

# CS 341

## Dr. Baker

### Fall 200?

For late-breaking information and news related to the class, see  
<http://www.cs.utexas.edu/users/dbaker/cs341/>

**Acknowledgements:**

This material was assembled by Elaine Rich, who allowed me to use it wholesale. I am greatly indebted to her for her efforts.

# Table of Contents

## I. Lecture Notes

### **A. Overview and Introduction**

1. The Three Hour Tour Through Automata Theory
2. What is a Language?

### **B. Regular Languages and Finite State Machines**

3. Regular Languages
4. Finite State Machines
5. Nondeterministic Finite State Machines
6. Interpreters for Finite State Machines
7. Equivalence of Regular Languages and FSMs
8. Languages that Are and Are Not Regular
9. A Review of Equivalence Relations
10. State Minimization
11. Summary of Regular Languages and Finite State Machines

### **C. Context-Free Languages and Pushdown Automata**

12. Context Free Grammars
13. Parse Trees
14. Pushdown Automata
15. Pushdown Automata and Context-Free Languages
16. Grammars and Normal Forms
17. Top Down Parsing
18. Bottom Up Parsing
19. Languages that Are and Are Not Context Free

### **D. Recursively Enumerable Languages, Turing Machines, and Decidability**

20. Turing Machines
21. Computing with Turing Machines
22. Recursively Enumerable and Recursive Languages
23. Turing Machine Extensions
24. Problem Encoding, Turing Machine Encoding, and the Universal Turing Machine
25. Grammars and Turing Machines
26. Undecidability
27. Introduction to Complexity Theory

## II. Homework

### **A. Review**

1. Basic Techniques

### **B. Regular Languages and Finite State Machines**

2. Strings and Languages
3. Languages and Regular Expressions
4. Deterministic Finite Automata
5. Regular Expressions in UNIX
6. Nondeterministic Finite Automata
7. Review of Equivalence Relations
8. Finite Automata, Regular Expressions, and Regular Grammars
9. Languages that Are and Are Not Regular
10. State Minimization

### **C. Context-Free Languages and Pushdown Automata**

11. Context Free Grammars
12. Parse Trees
13. Pushdown Automata
14. Pushdown Automata and Context-Free Grammars
15. Parsing
16. Languages that Are and Are Not Context-Free

### **D. Recursively Enumerable Languages, Turing Machines, and Decidability**

17. Turing Machines
18. Computing with Turing Machines
19. Turing Machine Extensions
20. Unrestricted Grammars
21. Undecidability

### **E. Review**

22. Review

## III. Supplementary Materials

- The Three Hour Tour through Automata Theory
- Review of Mathematical Concepts
- Regular Languages and Finite State Machines
- Context-Free Languages and Pushdown Automata
- Recursively Enumerable Languages, Turing Machines, and Decidability

# I. Lecture Notes

# The Three Hour Tour Through Automata Theory

Read Supplementary Materials: The Three Hour Tour Through Automata Theory

Read Supplementary Materials: Review of Mathematical Concepts

Read K & S Chapter 1

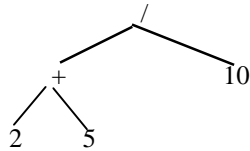
Do Homework 1.

## Let's Look at Some Problems

```
int alpha, beta;
alpha = 3;
beta = (2 + 5) / 10;
```

(1) **Lexical analysis:** Scan the program and break it up into variable names, numbers, etc.

(2) **Parsing:** Create a tree that corresponds to the sequence of operations that should be executed, e.g.,



(3) **Optimization:** Realize that we can skip the first assignment since the value is never used and that we can precompute the arithmetic expression, since it contains only constants.

(4) **Termination:** Decide whether the program is guaranteed to halt.

(5) **Interpretation:** Figure out what (if anything) it does.

## A Framework for Analyzing Problems

We need a single framework in which we can analyze a very diverse set of problems.

The framework we will use is **Language Recognition**

A *language* is a (possibly infinite) set of finite length strings over a finite alphabet.

## Languages

(1)  $\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$

L = {w  $\in \Sigma^*$ : w represents an odd integer}  
= {w  $\in \Sigma^*$ : the last character of w is 1,3,5,7, or 9}  
=  $(0 \cup 1 \cup 2 \cup 3 \cup 4 \cup 5 \cup 6 \cup 7 \cup 8 \cup 9)^* (1 \cup 3 \cup 5 \cup 7 \cup 9)$

(2)  $\Sigma = \{(\,)\}$

L = {w  $\in \Sigma^*$ : w has matched parentheses}  
= the set of strings accepted by the grammar:  
 $S \rightarrow ( S )$   
 $S \rightarrow SS$   
 $S \rightarrow \epsilon$

(3) L = {w: w is a sentence in English}

Examples: Mary hit the ball.  
Colorless green ideas sleep furiously.  
The window needs fixed.

(4) L = {w: w is a C program that halts on all inputs}

## Encoding Output in the Input String

(5) Encoding multiplication as a single input string

$L = \{w \text{ of the form: } \langle \text{integer} \rangle x \langle \text{integer} \rangle = \langle \text{integer} \rangle, \text{ where } \langle \text{integer} \rangle \text{ is any well formed integer, and the third integer is the product of the first two}\}$

12x9=108                      12=12                      12x8=108

(6) Encoding prime decomposition

$L = \{w \text{ of the form: } \langle \text{integer}1 \rangle / \langle \text{integer}2 \rangle, \langle \text{integer}3 \rangle \dots, \text{ where integers } 2 - n \text{ represent the prime decomposition of integer } 1.\}$

15/3,5                              2/2

## More Languages

(7) Sorting as a language recognition task:

$L = \{w_1 \# w_2: \exists n \geq 1, \text{ } w_1 \text{ is of the form } int_1, int_2, \dots int_n, \text{ } w_2 \text{ is of the form } int_1, int_2, \dots int_n, \text{ and } w_2 \text{ contains the same objects as } w_1 \text{ and } w_2 \text{ is sorted}\}$

Examples:

1,5,3,9,6#1,3,5,6,9  $\in L$

1,5,3,9,6#1,2,3,4,5,6,7  $\notin L$

(8) Database querying as a language recognition task:

$L = \{d \# q \# a: \text{ } d \text{ is an encoding of a database, } q \text{ is a string representing a query, and } a \text{ is the correct result of applying } q \text{ to } d\}$

Example:

(name, age, phone), (John, 23, 567-1234) (Mary, 24, 234-9876 )# (select name age=23) # (John)  $\in L$

## The Traditional Problems and their Language Formulations are Equivalent

By equivalent we mean:

If we have a machine to solve one, we can use it to build a machine to do the other using just the starting machine and other functions that can be built using a machine of equal or lesser power.

Consider the multiplication example:

$L = \{w \text{ of the form: } \langle \text{integer} \rangle x \langle \text{integer} \rangle = \langle \text{integer} \rangle, \text{ where } \langle \text{integer} \rangle \text{ is any well formed integer, and the third integer is the product of the first two}\}$

Given a multiplication machine, we can build the language recognition machine:

Given the language recognition machine, we can build a multiplication machine:

## A Framework for Describing Languages

Clearly, if we are going to work with languages, each one must have a finite description.

Finite Languages: Easy. Just list the elements of the language.

$L = \{\text{June, July, August}\}$

Infinite Languages: Need a finite description.

Grammars let us use recursion to do this.

### Grammars 1

(1) The Language of Matched Parentheses

$S \rightarrow (S)$   
 $S \rightarrow SS$   
 $S \rightarrow \epsilon$

(2) The Language of Odd Integers

$S \rightarrow 1$   
 $S \rightarrow 3$   
 $S \rightarrow 5$   
 $S \rightarrow 7$   
 $S \rightarrow 9$   
 $S \rightarrow 0S$   
 $S \rightarrow 1S$   
 $S \rightarrow 2S$   
 $S \rightarrow 3S$   
 $S \rightarrow 4S$   
 $S \rightarrow 5S$   
 $S \rightarrow 6S$   
 $S \rightarrow 7S$   
 $S \rightarrow 8S$   
 $S \rightarrow 9S$

### Grammars 2

$S \rightarrow O$   
 $S \rightarrow AO$   
 $A \rightarrow AD$   
 $A \rightarrow D$   
 $D \rightarrow O$   
 $D \rightarrow E$   
 $O \rightarrow 1$   
 $O \rightarrow 3$   
 $O \rightarrow 5$   
 $O \rightarrow 7$   
 $O \rightarrow 9$   
 $E \rightarrow 0$   
 $E \rightarrow 2$   
 $E \rightarrow 4$   
 $E \rightarrow 6$   
 $E \rightarrow 8$

### Grammars 3

(3) The Language of Simple Arithmetic Expressions

$S \rightarrow \langle \text{exp} \rangle$   
 $\langle \text{exp} \rangle \rightarrow \langle \text{number} \rangle$   
 $\langle \text{exp} \rangle \rightarrow (\langle \text{exp} \rangle)$   
 $\langle \text{exp} \rangle \rightarrow - \langle \text{exp} \rangle$   
 $\langle \text{exp} \rangle \rightarrow \langle \text{exp} \rangle \langle \text{op} \rangle \langle \text{exp} \rangle$   
 $\langle \text{op} \rangle \rightarrow + | - | * | /$   
 $\langle \text{number} \rangle \rightarrow \langle \text{digit} \rangle$   
 $\langle \text{number} \rangle \rightarrow \langle \text{digit} \rangle \langle \text{number} \rangle$   
 $\langle \text{digit} \rangle \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$



## Grammars as Generators and Acceptors

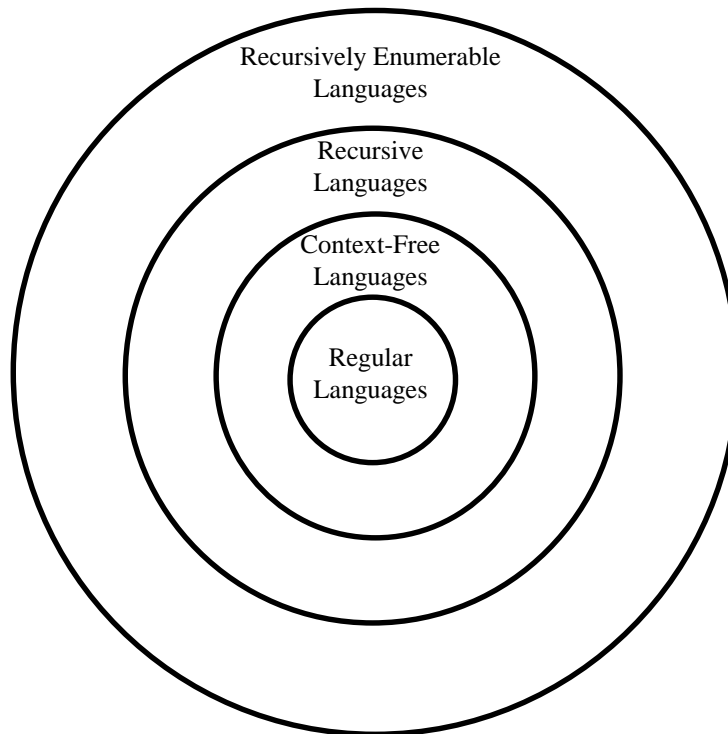
### Top Down Parsing

4 + 3

### Bottom Up Parsing

4 + 3

### The Language Hierarchy



## Regular Grammars

In a regular grammar, all rules must be of the form:

$\langle \text{one nonterminal} \rangle \rightarrow \langle \text{one terminal} \rangle \text{ or } \epsilon$

or

$\langle \text{one nonterminal} \rangle \rightarrow \langle \text{one terminal} \rangle \langle \text{one nonterminal} \rangle$

So, the following rules are okay:

$S \rightarrow \epsilon$

$S \rightarrow a$

$S \rightarrow aS$

But these are not:

$S \rightarrow ab$

$S \rightarrow SS$

$aS \rightarrow b$

## Regular Expressions and Languages

Regular expressions are formed from  $\emptyset$  and the characters in the target alphabet, plus the operations of:

- Concatenation:  $\alpha\beta$  means  $\alpha$  followed by  $\beta$
- Or (Set Union):  $\alpha\cup\beta$  means  $\alpha$  Or (Union)  $\beta$
- Kleene \*:  $\alpha^*$  means 0 or more occurrences of  $\alpha$  concatenated together.
- At Least 1:  $\alpha^+$  means 1 or more occurrences of  $\alpha$  concatenated together.
- $()$ : used to group the other operators

Examples:

(1) Odd integers:

$(0\cup 1\cup 2\cup 3\cup 4\cup 5\cup 6\cup 7\cup 8\cup 9)^*(1\cup 3\cup 5\cup 7\cup 9)$

(2) Identifiers:

$(A-Z)^+((A-Z) \cup (0-9))^*$

(3) Matched Parentheses

## Context Free Grammars

(1) The Language of Matched Parentheses

$S \rightarrow ( S )$

$S \rightarrow SS$

$S \rightarrow \epsilon$

(2) The Language of Simple Arithmetic Expressions

$S \rightarrow \langle \text{exp} \rangle$

$\langle \text{exp} \rangle \rightarrow \langle \text{number} \rangle$

$\langle \text{exp} \rangle \rightarrow (\langle \text{exp} \rangle)$

$\langle \text{exp} \rangle \rightarrow - \langle \text{exp} \rangle$

$\langle \text{exp} \rangle \rightarrow \langle \text{exp} \rangle \langle \text{op} \rangle \langle \text{exp} \rangle$

$\langle \text{op} \rangle \rightarrow + | - | * | /$

$\langle \text{number} \rangle \rightarrow \langle \text{digit} \rangle$

$\langle \text{number} \rangle \rightarrow \langle \text{digit} \rangle \langle \text{number} \rangle$

$\langle \text{digit} \rangle \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$

## Not All Languages are Context-Free

**English:**  $S \rightarrow NP VP$   
 $NP \rightarrow the NP1 | NP1$   
 $NP1 \rightarrow ADJ NP1 | N$   
 $N \rightarrow boy | boys$   
 $VP \rightarrow V | V NP$   
 $V \rightarrow run | runs$   
What about “boys runs”

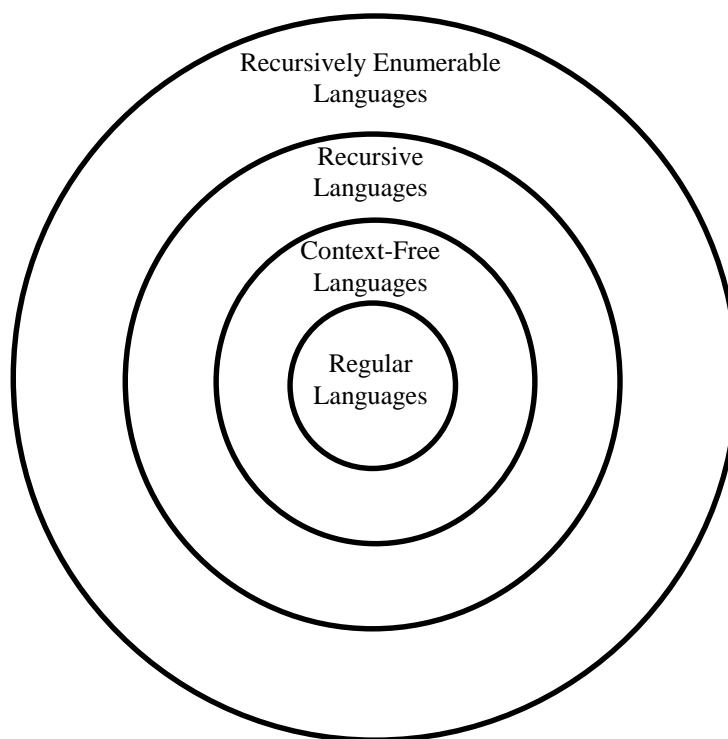
**A much simpler example:**  $a^n b^n c^n, n \geq 1$

## Unrestricted Grammars

Example: A grammar to generate all strings of the form  $a^n b^n c^n, n \geq 1$

$S \rightarrow aBSc$   
 $S \rightarrow aBc$   
 $Ba \rightarrow aB$   
 $Bc \rightarrow bc$   
 $Bb \rightarrow bb$

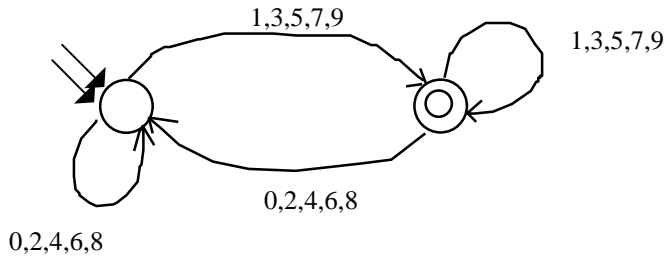
## The Language Hierarchy



## A Machine Hierarchy

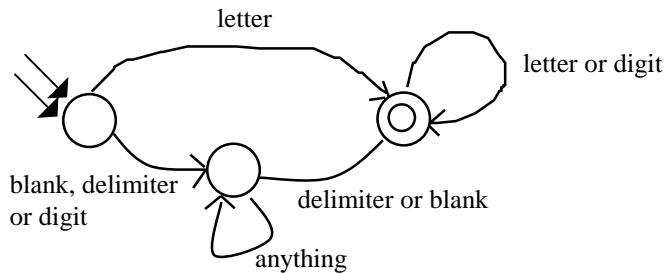
### Finite State Machines 1

An FSM to accept odd integers:



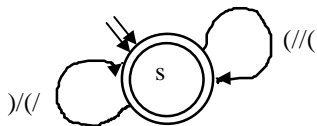
### Finite State Machines 2

An FSM to accept identifiers:



### Pushdown Automata

A PDA to accept strings with balanced parentheses:

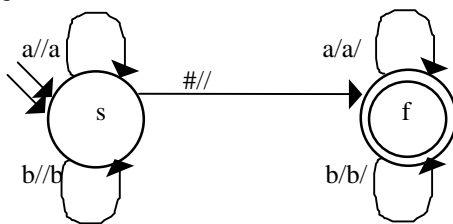


Example: (())()

Stack:

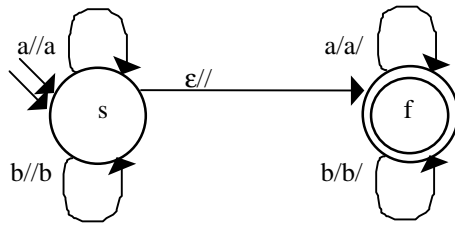
### Pushdown Automaton 2

A PDA to accept strings of the form  $w#w^R$ :



### A Nondeterministic PDA

A PDA to accept strings of the form  $ww^R$

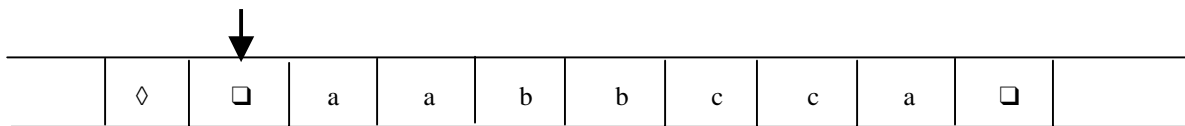
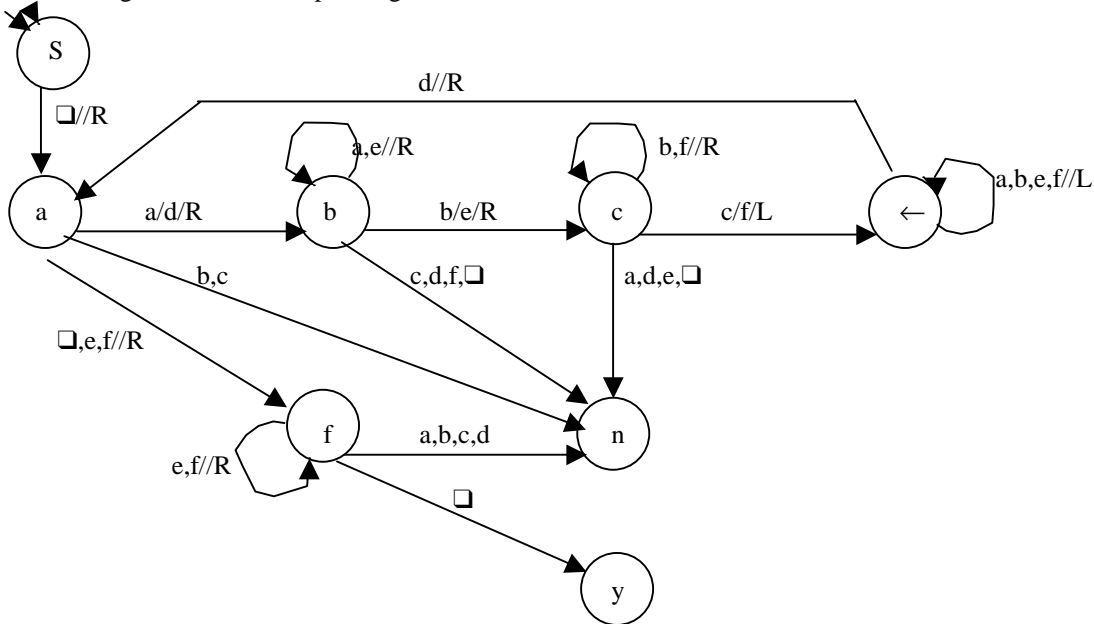


### PDA 3

A PDA to accept strings of the form  $a^n b^n c^n$

### Turing Machines

A Turing Machine to accept strings of the form  $a^n b^n c^n$

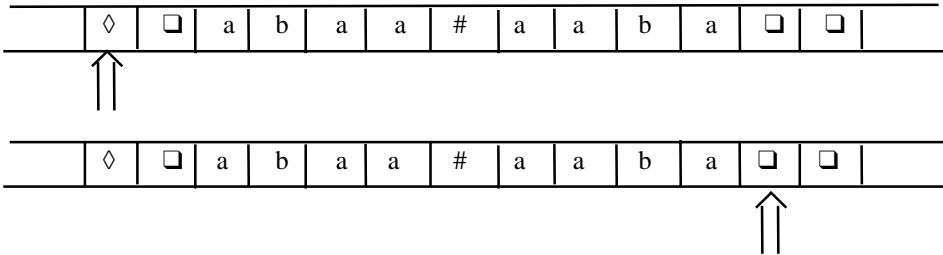


### A Two Tape Turing Machine

A Turing Machine to accept  $\{w#w^R\}$



A Two Tape Turing Machine to do the same thing



### Simulating k Tapes with One

A multitrack tape:

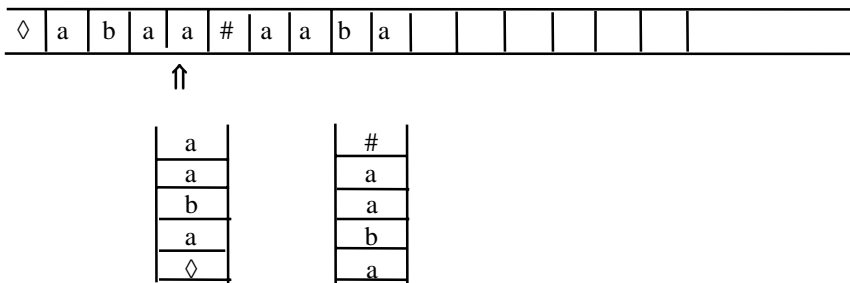
◇	◇	□	a	b	a	□	□	□	□
	0	0	1	0	0	0	0		
	◇	a	b	b	a	b	a		
	0	1	0	0	0	0	0		

Can be encoded on a single tape with an alphabet consisting of symbols corresponding to :

$$\{\{\diamond, a, b, \#, \square\} \times \{0, 1\} \times \{\diamond, a, b, \#, \square\} \times \{0, 1\}\}$$

Example: 2nd square:  $(\square, 0, a, 1)$

### Simulating a Turing Machine with a PDA with Two Stacks



## The Universal Turing Machine Encoding States, Symbols, and Transitions

Suppose the input machine  $M$  has 5 states, 4 tape symbols, and a transition of the form:

$(s,a,q,b)$ , which can be read as:

in state  $s$ , reading an  $a$ , go to state  $q$ , and write  $b$ .

We encode this transition as:

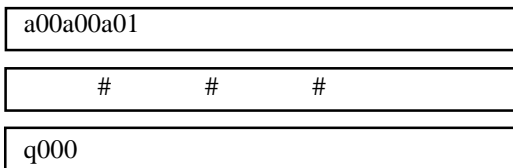
$q000,a00,q010,a01$

A series of transitions that describe an entire machine will look like

$q000,a00,q010,a01\#q010,a00,q000,a00$

### The Universal Turing Machine

$a \quad a \quad b$



### Church's Thesis (Church-Turing Thesis)

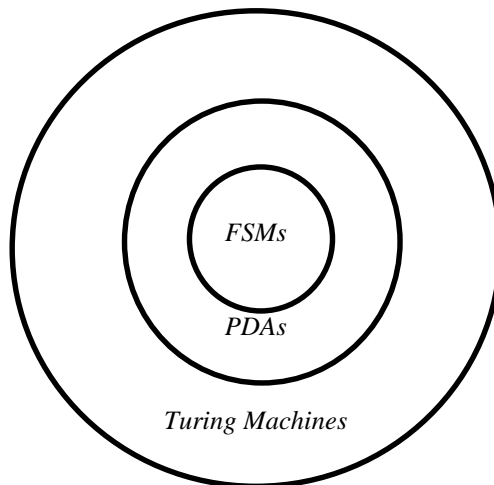
An algorithm is a formal procedure that halts.

The Thesis: Anything that can be computed by any algorithm can be computed by a Turing machine.

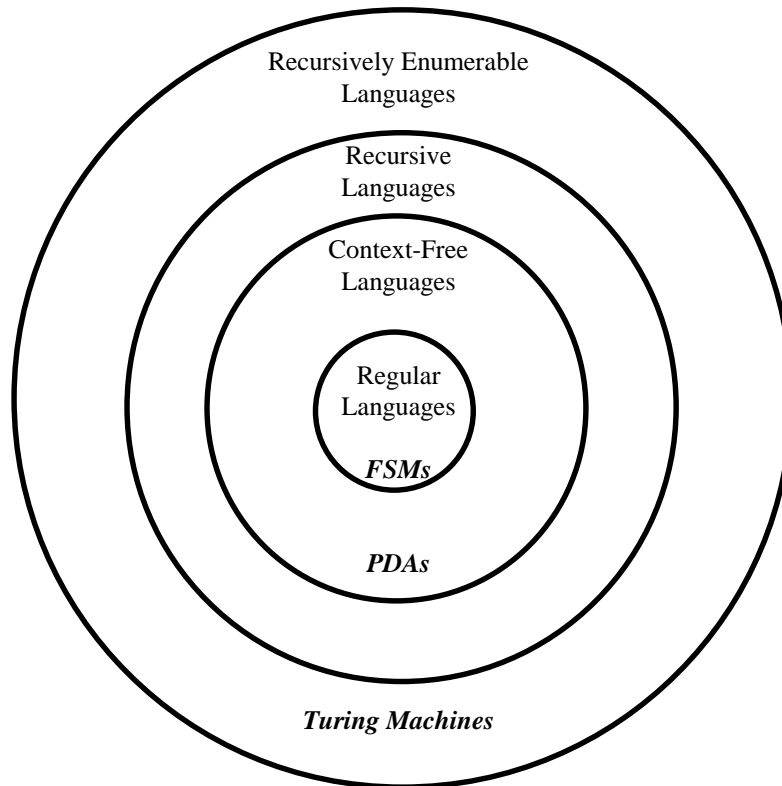
Another way to state it: All "reasonable" formal models of computation are equivalent to the Turing machine. This isn't a formal statement, so we can't prove it. But many different computational models have been proposed and they all turn out to be equivalent.

Example: unrestricted grammars

### A Machine Hierarchy



## Languages and Machines



### Where Does a Particular Problem Go?

Showing what it is -- generally by construction of:

- A grammar, or a machine

Showing what it isn't -- generally by contradiction, using:

- Counting  
Example:  $a^n b^n$
- Closure properties
- Diagonalization
- Reduction

### Closure Properties

#### Regular Languages are Closed Under:

- Union
- Concatenation
- Kleene closure
- Complementation
- Reversal
- Intersection

#### Context Free Languages are Closed Under:

- Union
- Concatenation
- Kleene Closure
- Reversal
- Intersection with regular languages

Etc.



## Using Closure Properties

Example:

$L = \{a^n b^m c^p : n \neq m \text{ or } m \neq p\}$  is not deterministic context-free.

Two theorems we'll prove later:

**Theorem 3.7.1:** The class of deterministic context-free languages is closed under complement.

**Theorem 3.5.2:** The intersection of a context-free language with a regular language is a context-free language.

If  $L$  were a deterministic CFL, then the complement of  $L$  ( $L'$ ) would be a deterministic CFL.

But  $L' \cap a^* b^* c^* = \{a^n b^n c^n\}$ , which we know is not context-free, much less deterministic context-free. Thus a contradiction.

### Diagonalization

The power set of the integers is not countable.

Imagine that there were some enumeration:

	1	2	3	4	5
Set 1	1				
Set 2		1		1	
Set 3	1		1		
Set 4		1			
Set 5	1	1	1	1	1

But then we could create a new set

New Set				1	
---------	--	--	--	---	--

But this new set must necessarily be different from all the other sets in the supposedly complete enumeration. Yet it should be included. Thus a contradiction.

### More on Cantor

Of course, if we're going to enumerate, we probably want to do it very systematically, e.g.,

	1	2	3	4	5	6	7
Set 1	1						
Set 2		1					
Set 3	1	1					
Set 4			1				
Set 5	1		1				
Set 6		1	1				
Set 7	1	1	1				

Read the rows as bit vectors, but read them backwards. So Set 4 is 100. Notice that this is the binary encoding of 4. This enumeration will generate all **finite** sets of integers, and in fact the set of all finite sets of integers is countable. But when will it generate the set that contains all the integers except 1?

## The Unsolvability of the Halting Problem

Suppose we could implement

HALTS(M,x)

M: string representing a Turing Machine

x: string representing the input for M

If M(x) halts then True

else False

Then we could define

TROUBLE(x)

x: string

If HALTS(x,x) then loop forever

else halt

So now what happens if we invoke TROUBLE(TROUBLE), which invokes

HALTS(TROUBLE,TROUBLE)

If HALTS says that TROUBLE halts on itself then TROUBLE loops. If HALTS says that TROUBLE loops, then TROUBLE halts.

### Viewing the Halting Problem as Diagonalization

First we need an enumeration of the set of all Turing Machines. We'll just use lexicographic order of the encodings we used as inputs to the Universal Turing Machine. So now, what we claim is that HALTS can compute the following table, where 1 means the machine halts on the input:

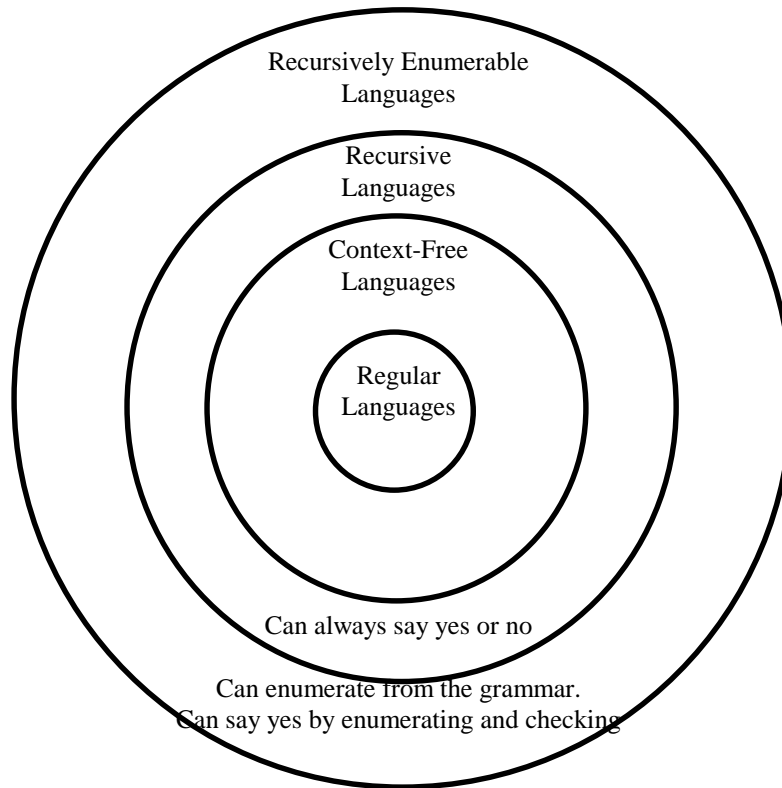
	I1	I2	I3	TROUBLE	I5
Machine 1	1				
Machine 2		1		1	
Machine 3					
TROUBLE			1		1
Machine 5	1	1	1	1	

But we've defined TROUBLE so that it will actually behave as:

TROUBLE			1	1	1
---------	--	--	---	---	---

Or maybe HALT said that TROUBLE(TROUBLE) would halt. But then TROUBLE would loop.

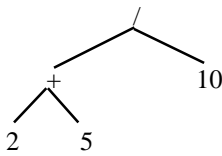
## Decidability



## Let's Revisit Some Problems

```
int alpha, beta;  
alpha = 3;  
beta = (2 + 5) / 10;
```

- (1) **Lexical analysis:** Scan the program and break it up into variable names, numbers, etc.
- (2) **Parsing:** Create a tree that corresponds to the sequence of operations that should be executed, e.g.,



- (3) **Optimization:** Realize that we can skip the first assignment since the value is never used and that we can precompute the arithmetic expression, since it contains only constants.
- (4) **Termination:** Decide whether the program is guaranteed to halt.
- (5) **Interpretation:** Figure out what (if anything) useful it does.

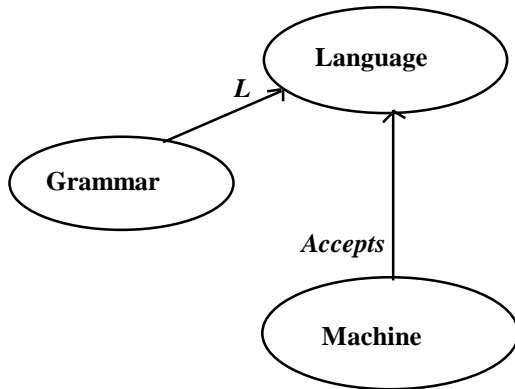
## So What's Left?

- Formalize and Prove Things
- Regular Languages and Finite State Machines
  - FSMs
    - Nondeterminism
    - State minimization
    - Implementation
  - Equivalence of regular expressions and FSMs
  - Properties of Regular Languages
- Context-Free Languages and PDAs
  - Equivalence of CFGs and nondeterministic PDAs
  - Properties of context-free languages
  - Parsing and determinism
- Turing Machines and Computability
  - Recursive and recursively enumerable languages
  - Extensions of Turing Machines
  - Undecidable problems for Turing Machines and unrestricted grammars

# What Is a Language?

Do Homework 2.

## Grammars, Languages, and Machines



## Strings: the Building Blocks of Languages

An **alphabet** is a finite set of **symbols**:

English alphabet: {A, B, C, ..., Z}

Binary alphabet: {0, 1}

A **string** over an alphabet is a finite sequence of symbols drawn from the alphabet.

English string: happynewyear

binary string: 1001101

We will generally omit “ ” from strings unless doing so would lead to confusion.

The set of all possible strings over an alphabet  $\Sigma$  is written  $\Sigma^*$ .

binary string:  $1001101 \in \{0,1\}^*$

The shortest string contains no characters. It is called the **empty string** and is written “ ” or  $\epsilon$  (epsilon).

The set of all possible strings over an alphabet  $\Sigma$  is written  $\Sigma^*$ .

## More on Strings

The **length** of a string is the number of symbols in it.

$|\epsilon| = 0$

$|1001101| = 7$

A string  $a$  is a **substring** of a string  $b$  if  $a$  occurs contiguously as part of  $b$ .

aaa is a substring of aaabbbaaa

aaaaaa is not a substring of aaabbbaaa

Every string is a substring (although not a proper substring) of itself.

$\epsilon$  is a substring of every string. Alternatively, we can match  $\epsilon$  anywhere.

Notice the analogy with sets here.

## Operations on Strings

**Concatenation:** The **concatenation** of two strings  $x$  and  $y$  is written  $x \parallel y$ ,  $x \cdot y$ , or  $xy$  and is the string formed by appending the string  $y$  to the string  $x$ .

$$|xy| = |x| + |y|$$

If  $x = \epsilon$  and  $y = \text{"food"}$ , then  $xy =$

If  $x = \text{"good"}$  and  $y = \text{"bye"}$ , then  $|xy| =$

**Note:**  $x \cdot \epsilon = \epsilon \cdot x = x$  for all strings  $x$ .

**Replication:** For each string  $w$  and each natural number  $i$ , the string  $w^i$  is defined recursively as

$$\begin{aligned} w^0 &= \epsilon \\ w^i &= w^{i-1} w \quad \text{for each } i \geq 1 \end{aligned}$$

Like exponentiation, the replication operator has a high precedence.

Examples:

$$\begin{aligned} a^3 &= \\ (\text{bye})^2 &= \\ a^0 b^3 &= \end{aligned}$$

## String Reversal

An inductive definition:

- (1) If  $|w| = 0$  then  $w^R = w = \epsilon$
- (2) If  $|w| \geq 1$  then  $\exists a \in \Sigma: w = u \cdot a$   
( $a$  is the last character of  $w$ )

and

$$w^R = a \cdot u^R$$

Example:

$$(\text{abc})^R =$$

## More on String Reversal

**Theorem:** If  $w, x$  are strings, then  $(w \cdot x)^R = x^R \cdot w^R$

$$\text{Example: } (\text{dogcat})^R = (\text{cat})^R \cdot (\text{dog})^R = \text{tacgod}$$

**Proof** (by induction on  $|x|$ ):

**Basis:**  $|x| = 0$ . Then  $x = \epsilon$ , and  $(w \cdot x)^R = (w \cdot \epsilon)^R = (w)^R = \epsilon \cdot w^R = \epsilon^R \cdot w^R = x^R \cdot w^R$

**Induction Hypothesis:** If  $|x| \leq n$ , then  $(w \cdot x)^R = x^R \cdot w^R$

**Induction Step:** Let  $|x| = n + 1$ . Then  $x = u \cdot a$  for some character  $a$  and  $|u| = n$

$$\begin{aligned} (w \cdot x)^R &= (w \cdot (u \cdot a))^R \\ &= ((w \cdot u) \cdot a)^R && \text{associativity} \\ &= a \cdot (w \cdot u)^R && \text{definition of reversal} \\ &= a \cdot u^R \cdot w^R && \text{induction hypothesis} \\ &= (u \cdot a)^R \cdot w^R && \text{definition of reversal} \\ &= x^R \cdot w^R \end{aligned}$$

$$\frac{\text{d o g c a t}}{w \quad \frac{x}{u \quad a}}$$

## Defining a Language

A **language** is a (finite or infinite) set of finite length strings over a finite alphabet  $\Sigma$ .

Example: Let  $\Sigma = \{a, b\}$

Some languages over  $\Sigma$ :  $\emptyset, \{\epsilon\}, \{a, b\}, \{\epsilon, a, aa, aaa, aaaa, aaaaa\}$

The language  $\Sigma^*$  contains an infinite number of strings, including:  $\epsilon, a, b, ab, ababaaa$

### Example Language Definitions

$L = \{x \in \{a, b\}^* : \text{all } a\text{'s precede all } b\text{'s}\}$

$L = \{x : \exists y \in \{a, b\}^* : x = ya\}$

$L = \{a^n, n \geq 0\}$

$L = a^n$  (If we say nothing about the range of  $n$ , we will assume that it is drawn from  $\mathbb{N}$ , i.e.,  $n \geq 0$ .)

$L = \{x\#y : x, y \in \{0-9\}^* \text{ and } \text{square}(x) = y\}$

$L = \{\} = \emptyset$  (the empty language—not to be confused with  $\{\epsilon\}$ , the language of the empty string)

### Techniques for Defining Languages

Languages are sets. Recall that, for sets, it makes sense to talk about *enumerations* and *decision procedures*. So, if we want to provide a computationally effective definition of a language we could specify either a

- Language **generator**, which enumerates (lists) the elements of the language, or a
- Language **recognizer**, which decides whether or not a candidate string is in the language and returns True if it is and False if it isn't.

Example: The logical definition:  $L = \{x : \exists y \in \{a, b\}^* : x = ya\}$  can be turned into either a language generator or a language recognizer.

### How Large are Languages?

- The smallest language over any alphabet is  $\emptyset$ .  $|\emptyset| = 0$
- The largest language over any alphabet is  $\Sigma^*$ .  $|\Sigma^*| = ?$ 
  - If  $\Sigma = \emptyset$  then  $\Sigma^* = \{\epsilon\}$  and  $|\Sigma^*| = 1$
  - If  $\Sigma \neq \emptyset$  then  $|\Sigma^*|$  is countably infinite because its elements can be enumerated in 1 to 1 correspondence with the integers as follows:
    1. Enumerate all strings of length 0, then length 1, then length 2, and so forth.
    2. Within the strings of a given length, enumerate them lexicographically. E.g., aa, ab, ba, bb
- So all languages are either finite or countably infinite. Alternatively, all languages are *countable*.

### Operations on Languages 1

Normal set operations: **union, intersection, difference, complement...**

Examples:  $\Sigma = \{a, b\}$   $L_1 = \text{strings with an even number of } a\text{'s}$   
 $L_2 = \text{strings with no } b\text{'s}$

$L_1 \cup L_2 =$

$L_1 \cap L_2 =$

$L_2 - L_1 =$

$\neg(L_2 - L_1) =$

## Operations on Languages 2

**Concatenation:** (based on the definition of concatenation of strings)

If  $L_1$  and  $L_2$  are languages over  $\Sigma$ , their concatenation  $L = L_1 L_2$ , sometimes  $L_1 \cdot L_2$ , is  
 $\{w \in \Sigma^* : w = xy \text{ for some } x \in L_1 \text{ and } y \in L_2\}$

Examples:

$L_1 = \{\text{cat, dog}\}$        $L_2 = \{\text{apple, pear}\}$        $L_1 L_2 = \{\text{catapple, catpear, dogapple, dogpear}\}$   
 $L_1 = \{a^n : n \geq 1\}$        $L_2 = \{a^n : n \leq 3\}$        $L_1 L_2 =$

**Identities:**

$L\emptyset = \emptyset L = \emptyset \quad \forall L$  (analogous to multiplication by 0)

$L\{\epsilon\} = \{\epsilon\}L = L \quad \forall L$  (analogous to multiplication by 1)

**Replicated concatenation:**

$L^n = L \cdot L \cdot L \cdot \dots \cdot L$  (n times)

$L^1 = L$

$L^0 = \{\epsilon\}$

Example:

$L = \{\text{dog, cat, fish}\}$

$L^0 = \{\epsilon\}$

$L^1 = \{\text{dog, cat, fish}\}$

$L^2 = \{\text{dogdog, dogcat, dogfish, catdog, catcat, catfish, fishdog, fishcat, fishfish}\}$

### Concatenating Languages Defined Using Variables

$L_1 = a^n = \{a^n : n \geq 0\}$

$L_2 = b^n = \{b^n : n \geq 0\}$

$L_1 L_2 = \{a^n : n \geq 0\} \{b^n : n \geq 0\} = \{a^n b^m : n, m \geq 0\}$  (common mistake: )  $\neq a^n b^n = \{a^n b^n : n \geq 0\}$

Note: The scope of any variable used in an expression that invokes replication will be taken to be the entire expression.

$L = 1^n 2^m$

$L = a^n b^m a^n$

## Operations on Languages 3

**Kleene Star (or Kleene closure):**  $L^* = \{w \in \Sigma^* : w = w_1 w_2 \dots w_k \text{ for some } k \geq 0 \text{ and some } w_1, w_2, \dots, w_k \in L\}$

Alternative definition:  $L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \dots$

Note:  $\forall L, \epsilon \in L^*$

Example:

$L = \{\text{dog, cat, fish}\}$

$L^* = \{\epsilon, \text{dog, cat, fish, dogdog, dogcat, fishcatfish, fishdogdogfishcat, ...}\}$

**Another useful definition:**  $L^+ = L L^*$  ( $L^+$  is the **closure** of  $L$  under concatenation)

Alternatively,  $L^+ = L^1 \cup L^2 \cup L^3 \cup \dots$

$L^+ = L^* - \{\epsilon\}$       if  $\epsilon \notin L$

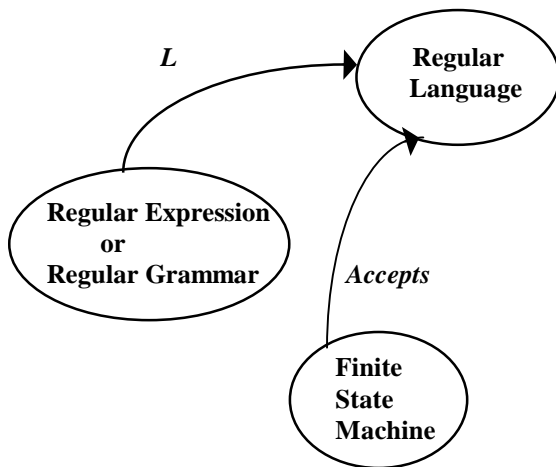
$L^+ = L^*$       if  $\epsilon \in L$



# Regular Languages

Read Supplementary Materials: Regular Languages and Finite State Machines: Regular Languages  
Do Homework 3.

## Regular Grammars, Languages, and Machines



## “Pure” Regular Expressions

The **regular expressions** over an alphabet  $\Sigma$  are all strings over the alphabet  $\Sigma \cup \{“(”, “)”, \emptyset, \cup, *\}$  that can be obtained as follows:

1.  $\emptyset$  and each member of  $\Sigma$  is a regular expression.
2. If  $\alpha, \beta$  are regular expressions, then so is  $\alpha\beta$
3. If  $\alpha, \beta$  are regular expressions, then so is  $\alpha\cup\beta$ .
4. If  $\alpha$  is a regular expression, then so is  $\alpha^*$ .
5. If  $\alpha$  is a regular expression, then so is  $(\alpha)$ .
6. Nothing else is a regular expression.

If  $\Sigma = \{a,b\}$  then these are regular expressions:  $\emptyset, a, bab, a\cup b, (a\cup b)^*a^*b^*$

So far, regular expressions are just (finite) strings over some alphabet,  $\Sigma \cup \{“(”, “)”, \emptyset, \cup, *\}$ .

## Regular Expressions Define Languages

Regular expressions define languages via a **semantic interpretation function** we'll call  $L$ :

1.  $L(\emptyset) = \emptyset$  and  $L(a) = \{a\}$  for each  $a \in \Sigma$
2. If  $\alpha, \beta$  are regular expressions, then
 
$$L(\alpha\beta) = L(\alpha) \cdot L(\beta)$$
 = all strings that can be formed by concatenating to some string from  $L(\alpha)$  some string from  $L(\beta)$ .  
 Note that if either  $\alpha$  or  $\beta$  is  $\emptyset$ , then its language is  $\emptyset$ , so there is nothing to concatenate and the result is  $\emptyset$ .
3. If  $\alpha, \beta$  are regular expressions, then  $L(\alpha\cup\beta) = L(\alpha) \cup L(\beta)$
4. If  $\alpha$  is a regular expression, then  $L(\alpha^*) = L(\alpha)^*$
5.  $L((\alpha)) = L(\alpha)$

A language is **regular** if and only if it can be described by a regular expression.

A regular expression is always finite, but it may describe a (countably) infinite language.

## Regular Languages

An equivalent definition of the class of regular languages over an alphabet  $\Sigma$ :

The **closure** of the languages

$$\{a\} \forall a \in \Sigma \text{ and } \emptyset \quad [1]$$

with respect to the functions:

- concatenation, [2]
- union, and [3]
- Kleene star. [4]

In other words, the class of regular languages is the smallest set that includes all elements of [1] and that is closed under [2], [3], and [4].

### “Closure” and “Closed”

Informally, a set can be defined in terms of a (usually small) starting set and a group of functions over elements from the set. The functions are applied to members of the set, and if anything new arises, it's added to the set. The resulting set is called the **closure** over the initial set and the functions. Note that the function(s) may only be applied a **finite** number of times.

Examples:

The set of natural numbers  $\mathbf{N}$  can be defined as the closure over  $\{0\}$  and the successor ( $\text{succ}(n) = n+1$ ) function.

Regular languages can be defined as the closure of  $\{a\} \forall a \in \Sigma$  and  $\emptyset$  and the functions of concatenation, union, and Kleene star.

We say a set is **closed** over a function if applying the function to arbitrary elements in the set does not yield any new elements.

Examples:

The set of natural numbers  $\mathbf{N}$  is closed under multiplication.

Regular languages are closed under intersection.

See *Supplementary Materials—Review of Mathematical Concepts* for more formal definitions of these terms.

### Examples of Regular Languages

$$L(a^*b^*) =$$

$$L(a \cup b) =$$

$$L((a \cup b)^*) =$$

$$L((a \cup b)^*a^*b^*) =$$

$$L = \{w \in \{a,b\}^* : |w| \text{ is even}\}$$

$$L = \{w \in \{a,b\}^* : w \text{ contains an odd number of } a\text{'s}\}$$

### Augmenting Our Notation

It would be really useful to be able to write  $\epsilon$  in a regular expression.

Example:  $(a \cup \epsilon) b$  (Optional a followed by b)

But we'd also like a minimal definition of what constitutes a regular expression. Why?

Observe that

$$\emptyset^0 = \{\epsilon\} \text{ (since 0 occurrences of the elements of any set generates the empty string), so}$$

$$\emptyset^* = \{\epsilon\}$$

So, without changing the set of languages that can be defined, we can add  $\epsilon$  to our notation for regular expressions if we specify that

$$L(\epsilon) = \{\epsilon\}$$

We're essentially treating  $\epsilon$  the same way that we treat the characters in the alphabet.

Having done this, you'll probably find that you rarely need  $\emptyset$  in any regular expression.

## More Regular Expression Examples

$L( (aa^*) \cup \epsilon ) =$   
 $L( (a \cup \epsilon)^* ) =$   
 $L = \{ w \in \{a,b\}^* : \text{there is no more than one } b \}$   
 $L = \{ w \in \{a,b\}^* : \text{no two consecutive letters are the same} \}$

### Further Notational Extensions of Regular Expressions



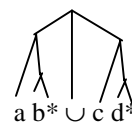
- A fixed number of concatenations:  $\alpha^n$  means  $\alpha\alpha\alpha\alpha\dots\alpha$  (n times).
- At Least 1:  $\alpha^+$  means 1 or more occurrences of  $\alpha$  concatenated together.
- Shorthands for denoting sets, such as ranges, e.g., (A-Z) or (letter-letter)  
 Example:  $L = (A-Z)^+((A-Z)\cup(0-9))^*$
- A replicated regular expression  $\alpha^n$ , where n is a constant.  
 Example:  $L = (0 \cup 1)^{20}$
- Intersection:  $\alpha \cap \beta$  (we'll prove later that regular languages are closed under intersection)  
 Example:  $L = (a^3)^* \cap (a^5)^*$

### Operator Precedence in Regular Expressions

Regular expressions are strings in the language of regular expressions. Thus to interpret them we need to:

1. Parse the string
2. Assign a meaning to the parse tree

Parsing regular expressions is a lot like parsing arithmetic expressions. To do it, we must assign precedence to the operators:

	<b>Regular Expressions</b>	<b>Arithmetic Expressions</b>
<b>Highest</b>	Kleene star	exponentiation
<div style="text-align: center;">  </div>	concatenation	multiplication
<div style="text-align: center;">  </div>	intersection	multiplication
<b>Lowest</b>	union	addition
		$x y^2 + i j^2$

### Regular Expressions and Grammars

Recall that grammars are language generators. A grammar is a recipe for creating strings in a language.

Regular expressions are analogous to grammars, but with two special properties:

1. They have limited power. They can be used to define only regular languages.
2. They don't look much like other kinds of grammars, which generally are composed of sets of production rules.

But we can write more "standard" grammars to define exactly the same languages that regular expressions can define. Specifically, any such grammar must be composed of rules that:

- have a left hand side that is a single nonterminal
- have a right hand side that is  $\epsilon$ , or a single terminal, or a single terminal followed by a single nonterminal.

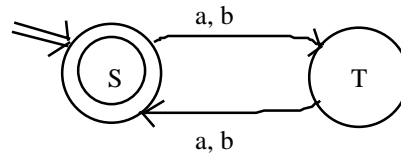
### Regular Grammar Example

$L = \{w \in \{a, b\}^* : |w| \text{ is even}\}$

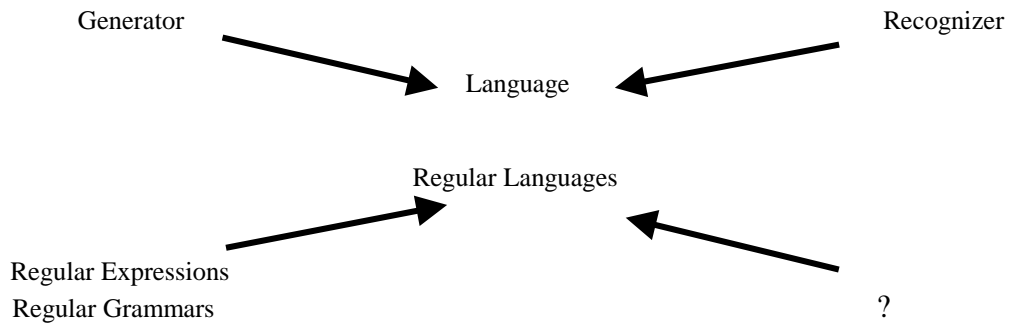
$((aa) \cup (ab) \cup (ba) \cup (bb))^*$

- $S \rightarrow \epsilon$
- $S \rightarrow aT$
- $S \rightarrow bT$
- $T \rightarrow a$
- $T \rightarrow b$
- $T \rightarrow aS$
- $T \rightarrow bS$

Notice how these rules correspond naturally to a FSM:



### Generators and Recognizers



# Finite State Machines

Read K & S 2.1  
Do Homeworks 4 & 5.

## Finite State Machines 1

A DFMS to accept odd integers:

### Definition of a Deterministic Finite State Machine (DFSM)

$M = (K, \Sigma, \delta, s, F)$ , where

- $K$  is a finite set of states
- $\Sigma$  is an alphabet
- $s \in K$  is the initial state
- $F \subseteq K$  is the set of final states, and
- $\delta$  is the transition function. It is function from  $(K \times \Sigma)$  to  $K$

i.e., each element of  $\delta$  maps from: a state, input symbol pair to a new state.

Informally,  $M$  **accepts** a string  $w$  if  $M$  winds up in some state that is an element of  $F$  when it has finished reading  $w$  (if not, it **rejects**  $w$ ).

The **language accepted by  $M$** , denoted  $L(M)$ , is the set of all strings accepted by  $M$ .

Deterministic finite state machines (DFSMs) are also called deterministic finite state automata (DFSAs or DFAs).

### Computations Using FSMs

A **computation** of A FSM is a sequence of configurations, where a configuration is any element of  $K \times \Sigma^*$ .

The **yields** relation  $\vdash_M$ :

- $(q, w) \vdash_M (q', w')$  iff
- $w = a w'$  for some symbol  $a \in \Sigma$ , and
  - $\delta(q, a) = q'$

(The yields relation effectively runs  $M$  one step.)

$\vdash_M^*$  is the reflexive, transitive closure of  $\vdash_M$ .

(The  $\vdash_M^*$  relation runs  $M$  any number of steps.)

Formally, a FSM  $M$  **accepts** a string  $w$  iff

$(s, w) \vdash_M^* (q, \epsilon)$ , for some  $q \in F$ .

### An Example Computation

A DFMS to accept odd integers:

On input 235, the configurations are:

$(q_0, 235) \quad \vdash_M \quad (q_0, 35)$   
 $\quad \quad \quad \vdash_M$   
 $\quad \quad \quad \vdash_M$

Thus  $(q_0, 235) \vdash_M^* (q_1, \epsilon)$ . (What does this mean?)

## Finite State Machines 2

A DFMSM to accept \$.50 in change:

### More Examples

$((aa) \cup (ab) \cup (ba) \cup (bb))^*$

$(b \cup \epsilon)(ab)^*(a \cup \epsilon)$

### More Examples

$L1 = \{w \in \{a, b\}^* : \text{every } a \text{ is immediately followed a } b\}$

A regular expression for L1:

A DFMSM for L1:

$L2 = \{w \in \{a, b\}^* : \text{every } a \text{ has a matching } b \text{ somewhere before it}\}$

A regular expression for L2:

A DFMSM for L2:

### Another Example: Socket-based Network Communication

<u>Client</u>	<u>Server</u>	$\Sigma = \{ \text{Open, Req, Reply, Close} \}$
open socket		
send request		
	send reply	$L = \text{Open (Req Reply)}^* (\text{Req} \cup \epsilon) \text{Close}$
send request		
	send reply	
...		$M =$
close socket		

### Definition of a Deterministic Finite State Transducer (DFST)

$M = (K, \Sigma, O, \delta, s, F)$ , where

$K$  is a finite set of states

$\Sigma$  is an input alphabet

$O$  is an output alphabet

$s \in K$  is the initial state

$F \subseteq K$  is the set of final states, and

$\delta$  is the transition function. It is function from

$(K \times \Sigma)$  to  $(K \times O^*)$

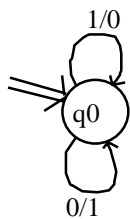
i.e., each element of  $\delta$  maps from: a state, input symbol pair  
to : a new state and zero or more output symbols (an output string)

$M$  computes a function  $M(w)$  if, when it reads  $w$ , it outputs  $M(w)$ .

Theorem: The output language of a deterministic finite state transducer (on final state) is regular.

### A Simple Finite State Transducer

Convert 1's to 0's and 0's to 1's (this isn't just a finite state task -- it's a one state task)



### An Odd Parity Generator

After every three bits, output a fourth bit such that each group of four bits has odd parity.

# Nondeterministic Finite State Machines

Read K & S 2.2, 2.3

Read Supplementary Materials: Regular Languages and Finite State Machines: Proof of the Equivalence of Nondeterministic and Deterministic FSAs.

Do Homework 6.

## Definition of a Nondeterministic Finite State Machine (NDFSM/NFA)

$M = (K, \Sigma, \Delta, s, F)$ , where

$K$  is a finite set of states

$\Sigma$  is an alphabet

$s \in K$  is the initial state

$F \subseteq K$  is the set of final states, and

$\Delta$  is the transition *relation*. It is a finite subset of

$$(K \times (\Sigma \cup \{\epsilon\})) \times K$$

i.e., each element of  $\Delta$  contains:

a configuration (state, input symbol or  $\epsilon$ ), and a new state.

$M$  accepts a string  $w$  if there exists *some path* along which  $w$  drives  $M$  to some element of  $F$ .

The language accepted by  $M$ , denoted  $L(M)$ , is the set of all strings accepted by  $M$ , where computation is defined analogously to DFMSs.

## A Nondeterministic FSA

$L = \{w : \text{there is a symbol } a_i \in \Sigma \text{ not appearing in } w\}$

The idea is to guess (nondeterministically) which character will be the one that doesn't appear.

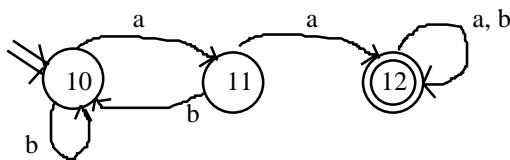
## Another Nondeterministic FSA

$L_1 = \{w : \text{aa occurs in } w\}$

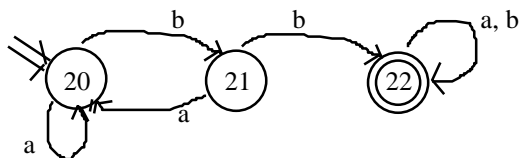
$L_2 = \{x : \text{bb occurs in } x\}$

$L_3 = \{y : y \in L_1 \text{ or } L_2\}$

$M_1 =$



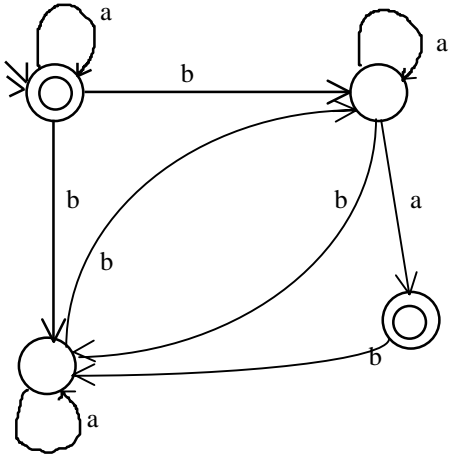
$M_2 =$



$M_3 =$



### Analyzing Nondeterministic FSAs

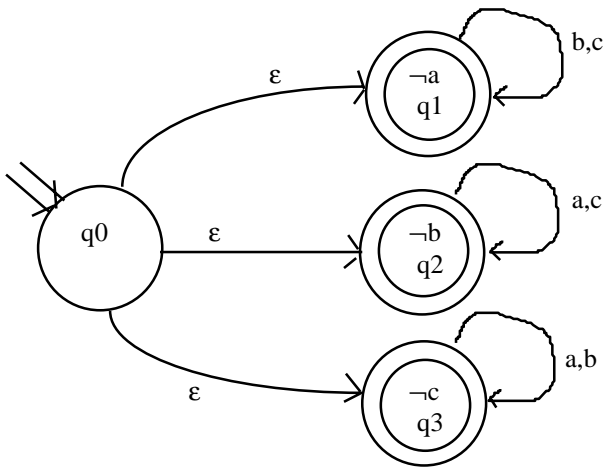


Does this FSA accept: baaba  
Remember: we just have to find one accepting path.

### Nondeterministic and Deterministic FSAs

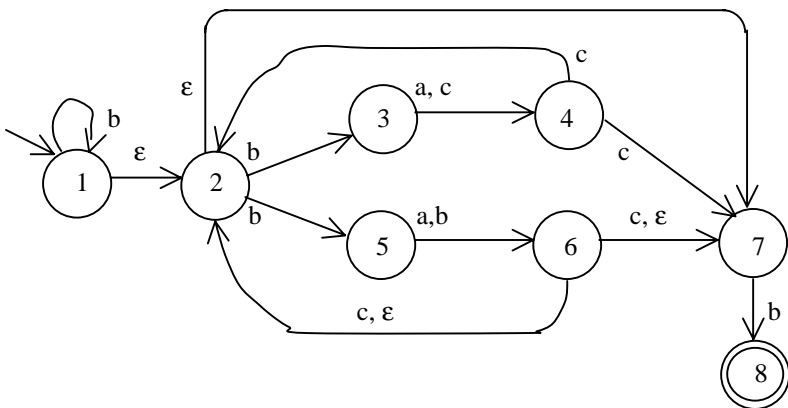
Clearly,  $\{\text{Languages accepted by a DFSA}\} \subseteq \{\text{Languages accepted by a NDFSA}\}$   
(Just treat  $\delta$  as  $\Delta$ )

More interestingly, **Theorem:** For each NDFSA, there is an equivalent DFSA.  
**Proof:** By construction

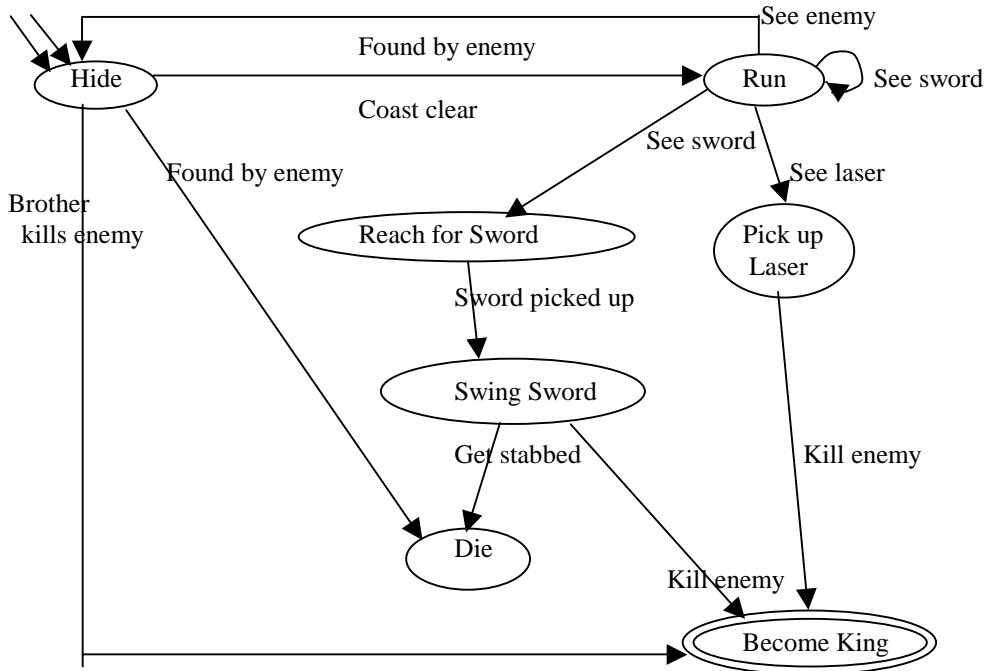


### Another Nondeterministic Example

$b^*(b(a \cup c)c \cup b(a \cup b)(c \cup \epsilon))^* b$



### A "Real" Example



### Dealing with $\epsilon$ Transitions

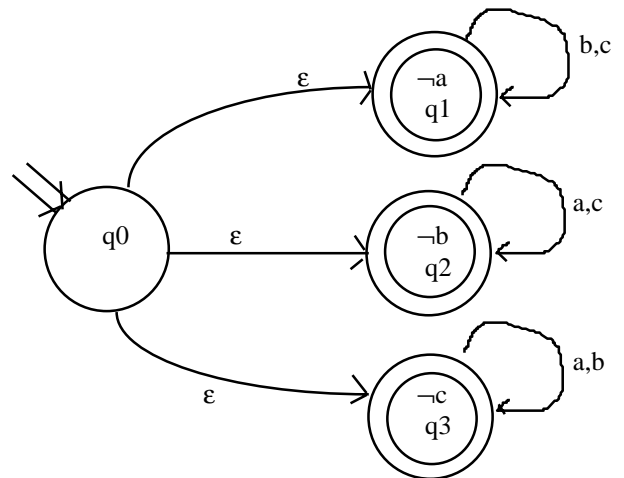
$E(q) = \{p \in K : (q,w) \vdash_M^* (p,w)\}$ .  $E(q)$  is the closure of  $\{q\}$  under the relation  $\{(p,r) : \text{there is a transition } (p, \epsilon, r) \in \Delta\}$   
 An algorithm to compute  $E(q)$ :

### Defining the Deterministic FSA

Given a NDFSA  $M = (K, \Sigma, \Delta, s, F)$ ,  
 we construct  $M' = (K', \Sigma, \delta', s', F')$ , where  
 $K' = 2^K$   
 $s' = E(s)$   
 $F' = \{Q \subseteq K : Q \cap F \neq \emptyset\}$   
 $\delta'(Q, a) = \cup \{E(p) : p \in K \text{ and } (q, a, p) \in \Delta \text{ for some } q \in Q\}$

Example: computing  $\delta'$  for the missing letter machine

$s' = \{q_0, q_1, q_2, q_3\}$   
 $\delta' = \{ (\{q_0, q_1, q_2, q_3\}, a, \{q_2, q_3\}), (\{q_0, q_1, q_2, q_3\}, b, \{q_1, q_3\}), (\{q_0, q_1, q_2, q_3\}, c, \{q_1, q_2\}), (\{q_1, q_2\}, a, \{q_2\}), (\{q_1, q_2\}, b, \{q_1\}), (\{q_1, q_2\}, c, \{q_1, q_2\}), (\{q_1, q_3\}, a, \{q_3\}), (\{q_1, q_3\}, b, \{q_1, q_3\}), (\{q_1, q_3\}, c, \{q_1\}), (\{q_2, q_3\}, a, \{q_2, q_3\}), (\{q_2, q_3\}, b, \{q_3\}), (\{q_2, q_3\}, c, \{q_2\}), (\{q_1\}, b, \{q_1\}), (\{q_1\}, c, \{q_1\}), (\{q_2\}, a, \{q_2\}), (\{q_2\}, c, \{q_2\}), (\{q_3\}, a, \{q_3\}), (\{q_3\}, b, \{q_3\}) \}$

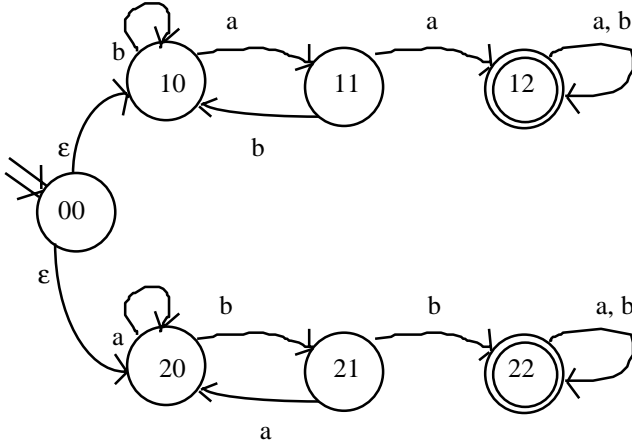


### An Algorithm for Constructing the Deterministic FSA

1. Compute the  $E(q)$ s:
2. Compute  $s' = E(s)$
3. Compute  $\delta'$ :  
 $\delta'(Q, a) = \cup \{E(p) : p \in K \text{ and } (q, a, p) \in \Delta \text{ for some } q \in Q\}$
4. Compute  $K' =$  a subset of  $2^K$
5. Compute  $F' = \{Q \in K' : Q \cap F \neq \emptyset\}$

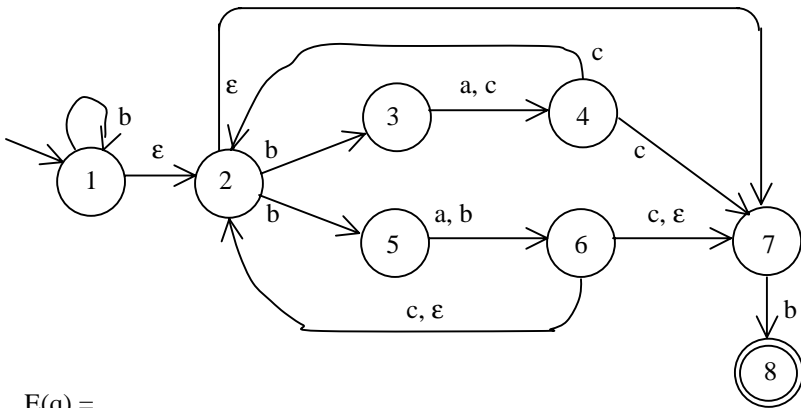
### An Example - The Or Machine

- $L_1 = \{w : \text{aa occurs in } w\}$   
 $L_2 = \{x : \text{bb occurs in } x\}$   
 $L_3 = \{y : \in L_1 \text{ or } L_2\}$



### Another Example

$$b^* (b(a \cup c)c \cup b(a \cup b) (c \cup \epsilon))^* b$$



$E(q) =$

$\delta' =$

## Sometimes the Number of States Grows Exponentially

Example: The missing letter machine, with  $|\Sigma| = n$

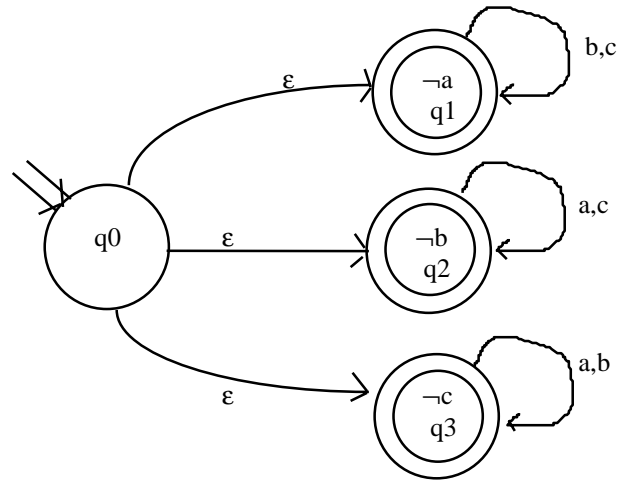
No. of states after 0 chars: 1

No. of new states after 1 char:  $\binom{n}{n-1} = n$

No. of new states after 2 chars:  $\binom{n}{n-2} = n(n-1)/2$

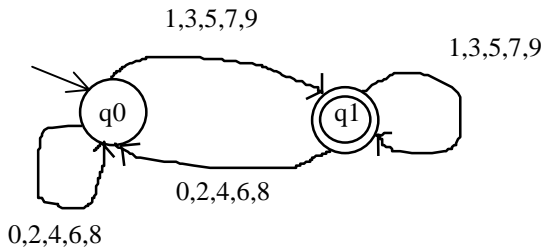
No. of new states after 3 chars:  $\binom{n}{n-3} = n(n-1)(n-2)/6$

Total number of states after n chars:  $2^n$



## What If The Original FSA is Deterministic?

M=



1. Compute the  $E(q)$ s:
2.  $s' = E(q_0) =$
3. Compute  $\delta'$ 
  - $(\{q_0\}, \text{odd}, \{q_1\})$
  - $(\{q_0\}, \text{even}, \{q_0\})$
  - $(\{q_1\}, \text{odd}, \{q_1\})$
  - $(\{q_1\}, \text{even}, \{q_0\})$
4.  $K' = \{\{q_0\}, \{q_1\}\}$
5.  $F' = \{\{q_1\}\}$

$$M' = M$$

## The real meaning of “determinism”

A FSA is **deterministic** if, for each input and state, there is at most one possible transition.

DFSAs are always deterministic. Why?

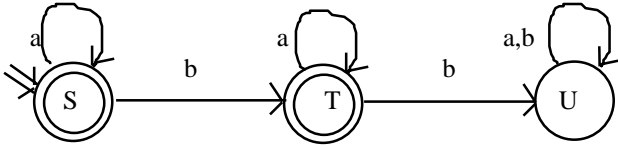
NFSAs can be deterministic (even with  $\epsilon$ -transitions and implicit dead states), but the formalism allows nondeterminism, in general.

Determinism implies uniquely defined machine behavior.

# Interpreters for Finite State Machines

## Deterministic FSAs as Algorithms

Example: No more than one b



Length of Program:  $|K| \times (|\Sigma| + 2)$

Time required to analyze string  $w$ :  $O(|w| \times |\Sigma|)$

We have to write new code for every new FSM.

Until accept or reject do:

```

S:  s := get-next-symbol;
    if s = end-of-file then accept;
    else if s = a then go to S;
    else if s = b then go to T;
T:  s := get-next-symbol;
    if s = end-of-file then accept;
    else if s = a then go to T;
    else if s = b then go to U;
etc.
  
```

## A Deterministic FSA Interpreter

To simulate  $M = (K, \Sigma, \delta, s, F)$ :

Simulate the no more than one b machine on input: aabaa

```

ST := s;
Repeat
  i := get-next-symbol;
  if i ≠ end-of-string then
    ST := δ(ST, i)
Until i = end-of-string;
If ST ∈ F then accept else reject
  
```

## Nondeterministic FSAs as Algorithms

Real computers are deterministic, so we have three choices if we want to execute a nondeterministic FSA:

1. Convert the NDFSA to a deterministic one:
  - Conversion can take time and space  $2^K$ .
  - Time to analyze string  $w$ :  $O(|w|)$
2. Simulate the behavior of the nondeterministic one by constructing sets of states "on the fly" during execution
  - No conversion cost
  - Time to analyze string  $w$ :  $O(|w| \times K^2)$
3. Do a depth-first search of all paths through the nondeterministic machine.

### A Nondeterministic FSA Interpreter

To simulate  $M = (K, \Sigma, \Delta, s, F)$ :

```
SET ST;
ST := E(s);
Repeat
  i := get-next-symbol;
  if i ≠ end-of-string then
    ST1 := ∅
    For all q ∈ ST do
      For all r ∈ Δ(q, i) do
        ST1 := ST1 ∪ E(r);
    ST := ST1;
Until i = end-of-string;
If  $ST \cap F \neq \emptyset$  then accept else reject
```

### A Deterministic Finite State Transducer Interpreter

To simulate  $M = (K, \Sigma, O, \delta, s, F)$ , given that:

$\delta_1(\text{state}, \text{symbol})$  returns a single new state  
(i.e.,  $M$  is deterministic), and  
 $\delta_2(\text{state}, \text{symbol})$  returns an element of  $O^*$ , the  
string to be output.

```
ST := s;
Repeat:
  i := get-next-symbol;
  if i ≠ end-of-string then
    write( $\delta_2(ST, i)$ );
    ST :=  $\delta_1(ST, i)$ 
Until i = end-of-string;
If  $ST \in F$  then accept else reject
```

# Equivalence of Regular Languages and FSMs

Read K & S 2.4

Read Supplementary Materials: Regular Languages and Finite State Machines: Generating Regular Expressions from Finite State Machines.

Do Homework 8.

## Equivalence of Regular Languages and FSMs

**Theorem:** The set of languages expressible using regular expressions (the regular languages) equals the class of languages recognizable by finite state machines. Alternatively, a language is regular if and only if it is accepted by a finite state machine.

### Proof Strategies

Possible Proof Strategies for showing that two sets,  $a$  and  $b$  are equal (also for iff):

1. Start with  $a$  and apply valid transformation operators until  $b$  is produced.

Example:

Prove:

$$\begin{aligned} A \cap (B \cup C) &= (A \cap B) \cup (A \cap C) \\ A \cap (B \cup C) &= (B \cup C) \cap A && \text{commutativity} \\ &= (B \cap A) \cup (C \cap A) && \text{distributivity} \\ &= (A \cap B) \cup (A \cap C) && \text{commutativity} \end{aligned}$$

2. Do two separate proofs: (1)  $a \Rightarrow b$ , and (2)  $b \Rightarrow a$ , possibly using totally different techniques. In this case, we show first (by construction) that for every regular expression there is a corresponding FSM. Then we show, by induction on the number of states, that for every FSM, there is a corresponding regular expression.

### For Every Regular Expression There is a Corresponding FSM

We'll show this by construction.

Example:

$$a^*(b \cup \epsilon)a^*$$

### Review - Regular Expressions

The regular expressions over an alphabet  $\Sigma^*$  are all strings over the alphabet  $\Sigma \cup \{ (, ), \emptyset, \cup, * \}$  that can be obtained as follows:

1.  $\emptyset$  and each member of  $\Sigma$  is a regular expression.
2. If  $\alpha, \beta$  are regular expressions, then so is  $\alpha\beta$ .
3. If  $\alpha, \beta$  are regular expressions, then so is  $\alpha \cup \beta$ .
4. If  $\alpha$  is a regular expression, then so is  $\alpha^*$ .
5. If  $\alpha$  is a regular expression, then so is  $(\alpha)$ .
6. Nothing else is a regular expression.

We also allow  $\epsilon$  and  $\alpha^+$ , etc. but these are just shorthands for  $\emptyset^*$  and  $\alpha\alpha^*$ , etc. so they do not need to be considered for completeness.

## For Every Regular Expression There is a Corresponding FSM

Formalizing the Construction: The class of regular languages is the smallest class of languages that contains  $\emptyset$  and each of the singleton strings drawn from  $\Sigma$ , and that is closed under

- Union
- Concatenation, and
- Kleene star

Clearly we can construct an FSM for any finite language, and thus for  $\emptyset$  and all the singleton strings. If we could show that the class of languages accepted by FSMs is also closed under the operations of union, concatenation, and Kleene star, then we could recursively construct, for any regular expression, the corresponding FSM, starting with the singleton strings and building up the machine as required by the operations used to express the regular expression.

### FSMs for Primitive Regular Expressions

An FSM for  $\emptyset$ :

An FSM for  $\epsilon$  ( $\emptyset^*$ ):

An FSM for a single element of  $\Sigma$ :

### Closure of FSMs Under Union

To create a FSM that accepts the union of the languages accepted by machines M1 and M2:

1. Create a new start state, and, from it, add  $\epsilon$ -transitions to the start states of M1 and M2.

### Closure of FSMs Under Concatenation

To create a FSM that accepts the concatenation of the languages accepted by machines M1 and M2:

1. Start with M1.
2. From every final state of M1, create an  $\epsilon$ -transition to the start state of M2.
3. The final states are the final states of M2.



### Closure of FSMs Under Kleene Star

To create an FSM that accepts the Kleene star of the language accepted by machine M1:

1. Start with M1.
2. Create a new start state  $S_0$  and make it a final state (so that we can accept  $\epsilon$ ).
3. Create an  $\epsilon$ -transition from  $S_0$  to the start state of M1.
4. Create  $\epsilon$ -transitions from all of M1's final states back to its start state.
5. Make all of M1's final states final.

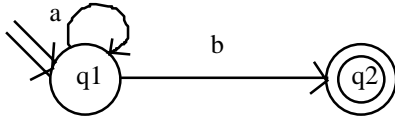
Note: we need a new start state,  $S_0$ , because the start state of the new machine must be a final state, and this may not be true of M1's start state.

### Closure of FSMs Under Complementation

To create an FSM that accepts the complement of the language accepted by machine M1:

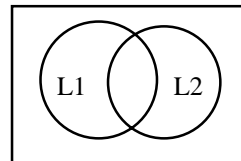
1. Make M1 deterministic.
2. Reverse final and nonfinal states.

#### A Complementation Example



### Closure of FSMs Under Intersection

$L_1 \cap L_2 =$



Write this in terms of operations we have already proved closure for:

- Union
- Concatenation
- Kleene star
- Complementation

#### An Example

$(b \cup ab^*a)^*ab^*$

## For Every FSM There is a Corresponding Regular Expression

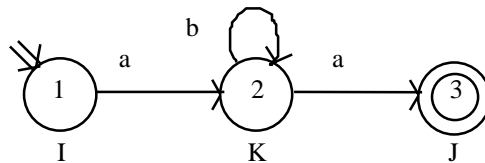
### Proof:

(1) There is a trivial regular expression that describes the strings that can be recognized in going from one state to itself ( $\{\epsilon\}$  plus any other single characters for which there are loops) or from one state to another directly (i.e., without passing through any other states), namely all the single characters for which there are transitions.

(2) Using (1) as the base case, we can build up a regular expression for an entire FSM by induction on the number assigned to possible intermediate states we can pass through. By adding them in only one at a time, we always get simple regular expressions, which can then be combined using union, concatenation, and Kleene star.

### Key Ideas in the Proof

**Idea 1:** Number the states and, at each induction step, increase by one the states that can serve as intermediate states.



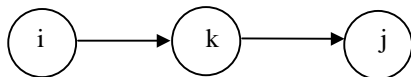
**Idea 2:** To get from state I to state J without passing through any intermediate state numbered greater than K, a machine may either:

1. Go from I to J without passing through any state numbered greater than K-1 (which we'll take as the induction hypothesis), or
2. Go from I to K, then from K to K any number of times, then from K to J, in each case without passing through any intermediate states numbered greater than K-1 (the induction hypothesis, again).

So we'll start with no intermediate states allowed, then add them in one at a time, each time building up the regular expression with operations under which regular languages are closed.

### The Formula

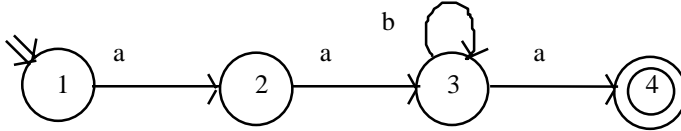
Adding in state k as an intermediate state we can use to go from i to j, described using paths that don't use k:



$$\begin{aligned}
 R(i, j, k) &= R(i, j, k-1) && /* \text{ what you could do without } k \\
 &\cup && \\
 R(i, k, k-1) &&& /* \text{ go from } i \text{ to the new intermediate state without using } k \text{ or higher} \\
 &\circ && \\
 R(k, k, k-1)^* &&& /* \text{ then go from the new intermediate state back to itself as many times as you want} \\
 &\circ && \\
 R(k, j, k-1) &&& /* \text{ then go from the new intermediate state to } j \text{ without using } k \text{ or higher}
 \end{aligned}$$

Solution:  $\cup R(s, q, N) \quad \forall q \in F$

### An Example of the Induction



Going through no intermediate states:

$$(1,1,0) = \epsilon \quad (1,2,0) = a \quad (1,3,0) = \emptyset \quad (2,3,0) = a \quad (3,3,0) = \epsilon \cup b \quad (3,4,0) = a$$

Allow 1 as an intermediate state:

Allow 2 as an intermediate state:

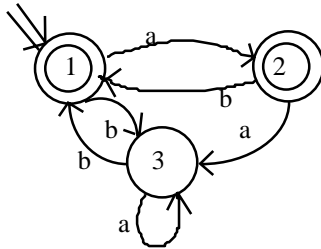
$$\begin{aligned} (1,3,2) &= (1,3,1) \cup (1,2,1)(2,2,1)^*(2,3,1) \\ &= \emptyset \cup a \epsilon^* a \\ &= aa \end{aligned}$$

Allow 3 as an intermediate state:

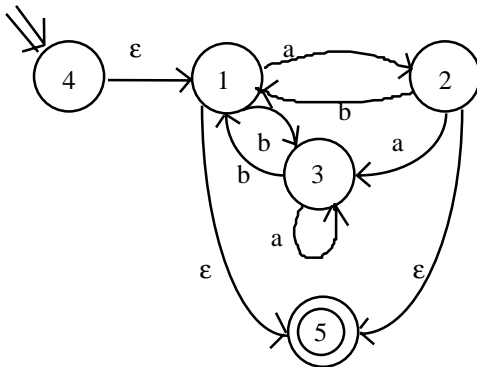
$$\begin{aligned} (1,3,3) &= (1,3,2) \cup (1,3,2)(3,3,2)^*(3,3,2) \\ &= aa \cup aa (\epsilon \cup b)^* (\epsilon \cup b) \\ &= aab^* \end{aligned}$$

$$\begin{aligned} (1,4,3) &= (1,4,2) \cup (1,3,2)(3,3,2)^*(3,4,2) \\ &= \emptyset \cup aa (\epsilon \cup b)^* a \\ &= aab^*a \end{aligned}$$

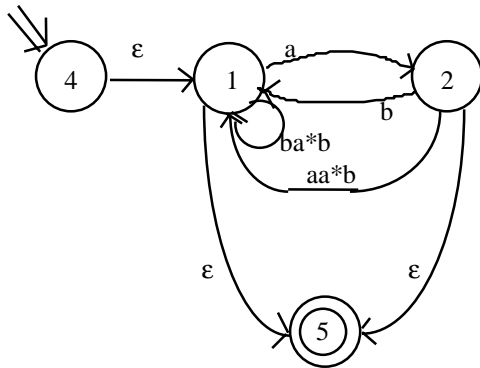
### An Easier Way - See Packet



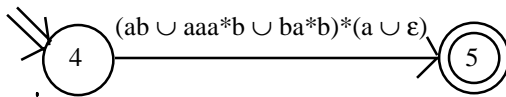
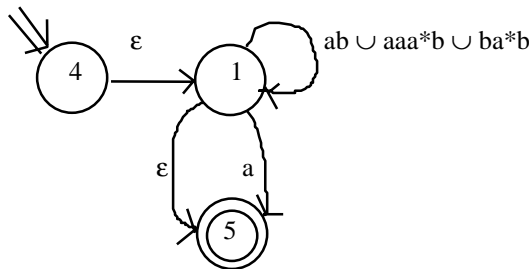
(1) Create a new initial state and a new, unique final state, neither of which is part of a loop.



(2) Remove states and arcs and replace with arcs labelled with larger and larger regular expressions. States can be removed in any order, but don't remove either the start or final state.



(Notice that the removal of state 3 resulted in two new paths because there were two incoming paths to 3 from another state and 1 outgoing path to another state, so  $2 \times 1 = 2$ .) The two paths from 2 to 1 should be coalesced by unioning their regular expressions (not shown).



Thus, the equivalent regular expression is:

$$(ab \cup aaa^*b \cup ba^*b)^*(a \cup \epsilon)$$

### Using Regular Expressions in the Real World (PERL)

#### Matching floating point numbers:

`-? ([0-9]+(\.[0-9]*)?) | \.[0-9]+`

#### Matching IP addresses:

`([0-9]+ (\.[0-9]+) {3})`

#### Finding doubled words:

`\< ([A-Za-z]+) \s+ \1 \>`

From Friedl, J., *Mastering Regular Expressions*, O'Reilly, 1997.

Note that some of these constructs are more powerful than regular expressions.

## Regular Grammars and Nondeterministic FSAs

Any regular language can be defined by a regular grammar, in which all rules

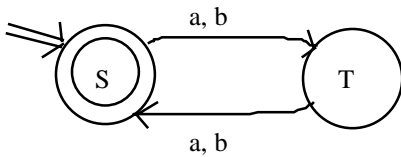
- have a left hand side that is a single nonterminal
- have a right hand side that is  $\epsilon$ , a single terminal, a single nonterminal, or a single terminal followed by a single nonterminal.

Example:  $L = \{w \in \{a, b\}^* : |w| \text{ is even}\}$

$$((aa) \cup (ab) \cup (ba) \cup (bb))^*$$

$$\begin{aligned} S &\rightarrow \epsilon \\ S &\rightarrow aT \\ S &\rightarrow bT \end{aligned}$$

$$\begin{aligned} T &\rightarrow a \\ T &\rightarrow b \\ T &\rightarrow aS \\ T &\rightarrow bS \end{aligned}$$



### An Algorithm to Generate the NDFSM from a Regular Grammar

1. Create a nonterminal for each state in the NDFSM.
2.  $s$  is the start state.
3. If there are any rules of the form  $X \rightarrow w$ , for some  $w \in \Sigma$ , then create an additional state labeled #.
4. For each rule of the form  $X \rightarrow w Y$ , add a transition from  $X$  to  $Y$  labeled  $w$  ( $w \in \Sigma \cup \epsilon$ ).
5. For each rule of the form  $X \rightarrow w$ , add a transition from  $X$  to # labeled  $w$  ( $w \in \Sigma$ ).
6. For each rule of the form  $X \rightarrow \epsilon$ , mark state  $X$  final.
7. Mark state # final.

#### Example 1 - Even Length Strings

$$\begin{aligned} S &\rightarrow \epsilon \\ S &\rightarrow aT \\ S &\rightarrow bT \end{aligned}$$

$$\begin{aligned} T &\rightarrow a \\ T &\rightarrow b \\ T &\rightarrow aS \\ T &\rightarrow bS \end{aligned}$$

#### Example 2 - One Character Missing

$$\begin{aligned} S &\rightarrow \epsilon \\ S &\rightarrow aB \\ S &\rightarrow aC \\ S &\rightarrow bA \\ S &\rightarrow bC \\ S &\rightarrow cA \\ S &\rightarrow cB \end{aligned}$$

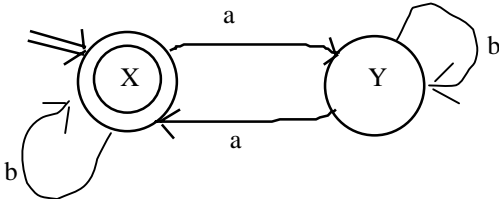
$$\begin{aligned} A &\rightarrow bA \\ A &\rightarrow cA \\ A &\rightarrow \epsilon \\ B &\rightarrow aB \\ B &\rightarrow cB \\ B &\rightarrow \epsilon \end{aligned}$$

$$\begin{aligned} C &\rightarrow aC \\ C &\rightarrow bC \\ C &\rightarrow \epsilon \end{aligned}$$

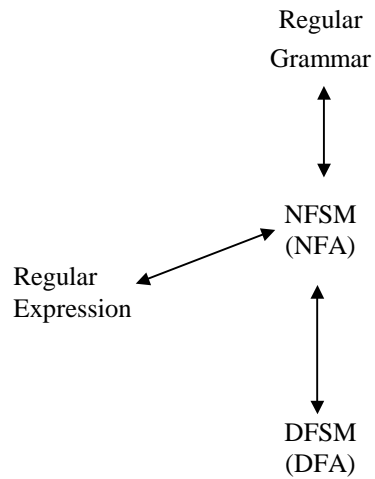
### An Algorithm to Generate a Regular Grammar from an NDFSM

1. Create a nonterminal for each state in the NDFSM.
2. The start state becomes the starting nonterminal
3. For each transition  $\delta(T, a) = U$ , make a rule of the form  $T \rightarrow aU$ .
4. For each final state  $T$ , make a rule of the form  $T \rightarrow \epsilon$ .

Example:



### Conversion Algorithms between Regular Language Formalisms



# Languages That Are and Are Not Regular

Read L & S 2.5, 2.6

Read Supplementary Materials: Regular Languages and Finite State Machines: The Pumping Lemma for Regular Languages.  
Do Homework 9.

## Deciding Whether a Language is Regular

**Theorem:** There exist languages that are not regular.

**Lemma:** There are an uncountable number of languages.

**Proof of Lemma:**

Let:  $\Sigma$  be a finite, nonempty alphabet, e.g.,  $\{a, b, c\}$ .

Then  $\Sigma^*$  contains all finite strings over  $\Sigma$ .

e.g.,  $\{\epsilon, a, b, c, aa, ab, bc, abc, bba, bbaa, bbbaac\}$

$\Sigma^*$  is countably infinite, because its elements can be enumerated one at a time, shortest first.

Any language  $L$  over  $\Sigma$  is a subset of  $\Sigma^*$ , e.g.,  $L1 = \{a, aa, aaa, aaaa, aaaaa, \dots\}$

$L2 = \{ab, abb, abbb, abbbb, abbbbbb, \dots\}$

The set of all possible languages is thus the power set of  $\Sigma^*$ .

The power set of any countably infinite set is not countable. So there are an uncountable number of languages over  $\Sigma^*$ .

## Some Languages Are Not Regular

**Theorem:** There exist languages that are not regular.

**Proof:**

(1) There are a countably infinite number of regular languages. This true because every description of a regular language is of finite length, so there is a countably infinite number of such descriptions.

(2) There are an uncountable number of languages.

Thus there are more languages than there are regular languages. So there must exist some language that is not regular.

## Showing That a Language is Regular

Techniques for showing that a language  $L$  is regular:

1. Show that  $L$  has a finite number of elements.
2. Exhibit a regular expression for  $L$ .
3. Exhibit a FSA for  $L$ .
4. Exhibit a regular grammar for  $L$ .
5. Describe  $L$  as a function of one or more other regular languages and the operators  $\cdot, \cup, \cap, *, -, \neg$ . We use here the fact that the regular languages are closed under all these operations.
6. Define additional operators and prove that the regular languages are closed under them. Then use these operators as in 5.

### Example

Let  $\Sigma = \{0, 1, 2, \dots, 9\}$

Let  $L \subseteq \Sigma^*$  be the set of decimal representations for nonnegative integers (with no leading 0's) divisible by 2 or 3.

$L_1 =$  decimal representations of nonnegative integers without leading 0's.

$$L_1 = 0 \cup \{1, 2, \dots, 9\}\{0-9\}^*$$

So  $L_1$  is regular.

$L_2 =$  decimal representations of nonnegative integers without leading 0's divisible by 2

$$L_2 = L_1 \cap \Sigma^*\{0, 2, 4, 6, 8\}$$

So  $L_2$  is regular.

### Example, Continued

$L_3 = L_1$  and divisible by 3

Recall that a number is divisible by 3 if and only if the sum of its digits is divisible by 3. We can build a FSM to determine that and accept the language  $L_{3a}$ , which is composed of strings of digits that sum to a multiple of 3.

$$L_3 = L_1 \cap L_{3a}$$

Finally,  $L = L_2 \cup L_3$

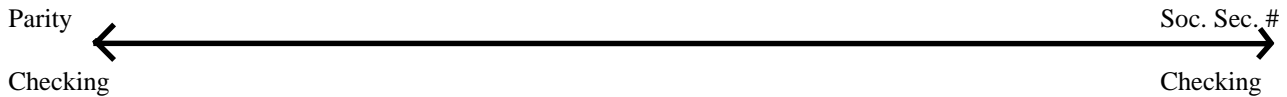
### Another Example

$\Sigma = \{0 - 9\}$

$L = \{w : w \text{ is the social security number of a living US resident}\}$

### Finiteness - Theoretical vs. Practical

Any finite language is regular. The size of the language doesn't matter.



But, from an implementation point of view, it very well may.

When is an FSA a good way to encode the facts about a language?

What are our alternatives?

FSA's are good at looking for repeating patterns. They don't bring much to the table when the language is just a set of unrelated strings.

### Showing that a Language is Not Regular

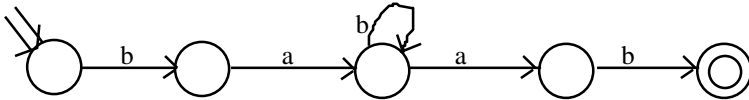
The argument, "I can't find a regular expression or a FSM", won't fly. (But a proof that there cannot exist a FSM is ok.)

Instead, we need to use two fundamental properties shared by regular languages:

1. We can only use a finite amount of memory to record essential properties.  
 Example:  
 $a^n b^n$  is not regular
2. The only way to generate/accept an infinite language with a finite description is to use Kleene star (in regular expressions) or cycles (in automata). This forces some kind of simple repetitive cycle within the strings.  
 Example:  
 $ab^*a$  generates aba, abba, abbba, abbbbba, etc.  
 Example:  
 $\{a^n : n \geq 1 \text{ is a prime number}\}$  is not regular.

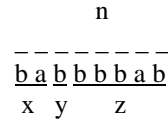


### Exploiting the Repetitive Property



If a FSM of  $n$  states accepts any string of length  $\geq n$ , how many strings does it accept?

$$L = bab^*ab$$



$xy^*z$  must be in  $L$ .

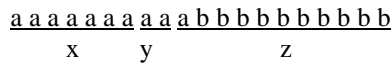
So  $L$  includes: baab, babab, babbab, babbabababab

### The Pumping Lemma for Regular Languages

If  $L$  is regular, then

- $\exists N \geq 1$ , such that
- $\forall$  strings  $w \in L$ , where  $|w| \geq N$ ,
- $\exists x, y, z$ , such that  $w = xyz$
- and  $|xy| \leq N$ ,
- and  $y \neq \epsilon$ ,
- and  $\forall q \geq 0$ ,  $xy^qz$  is in  $L$ .

Example:  $L = a^n b^n$

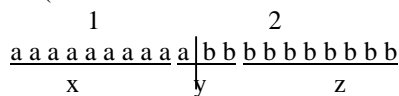


- $\exists N \geq 1$  Call it  $N$
- $\forall$  long strings  $w$  We pick one
- $\exists x, y, z$  We show no  $x, y, z$

### Example: $a^n b^n$ is not Regular

$N$  is the number from the pumping lemma (or one more, if  $N$  is odd).

Choose  $w = a^{N/2} b^{N/2}$ . (Since this is what it takes to be "long enough":  $|w| \geq N$ )



We show that there is no  $x, y, z$  with the required properties:

- $|xy| \leq N$ ,
- $y \neq \epsilon$ ,
- $\forall q \geq 0$ ,  $xy^qz$  is in  $L$ .

Three cases to consider:

- $y$  falls in region 1:
- $y$  falls across regions 1 and 2:
- $y$  falls in region 3:

**Example: a<sup>n</sup>b<sup>n</sup> is not Regular**

Second try:

Choose w to be a<sup>N</sup>b<sup>N</sup>. (Since we get to choose any w in L.)

$$\begin{array}{ccccccc}
& & 1 & & 2 & & \\
& & a & a & a & a & a & a & a & | & b & b & b & b & b & b & b & b & b & \\
& & x & & y & & z & & & & & & & & & & & & & & 
\end{array}$$

We show that there is no x, y, z with the required properties:

- |xy| ≤ N,
- y ≠ ε,
- ∀ q ≥ 0, xy<sup>q</sup>z is in L.

Since |xy| ≤ N, y must be in region 1. So y = a<sup>g</sup> for some g ≥ 1. Pumping in or out (any q but 1) will violate the constraint that the number of a’s has to equal the number of b’s.

**A Complete Proof Using the Pumping Lemma**

Proof that L = {a<sup>n</sup>b<sup>n</sup>} is not regular:

Suppose L is regular. Since L is regular, we can apply the pumping lemma to L. Let N be the number from the pumping lemma for L. Choose w = a<sup>N</sup>b<sup>N</sup>. Note that w ∈ L and |w| ≥ N. From the pumping lemma, there exists some x, y, z where xyz = w and |xy| ≤ N, y ≠ ε, and ∀ q ≥ 0, xy<sup>q</sup>z ∈ L. Because |xy| ≤ N, y = a<sup>|y|</sup> (y is all a’s). We choose q = 2 and xy<sup>q</sup>z = a<sup>N+|y|</sup>b<sup>N</sup>. Because |y| > 0, then xy<sup>2</sup>z ∉ L (the string has more a’s than b’s). Thus for all possible x, y, z: xyz = w, ∃q, xy<sup>q</sup>z ∉ L. Contradiction. ∴ L is not regular.

Note: the underlined parts of the above proof is “boilerplate” that can be reused. A complete proof should have this text or something equivalent.

You get to choose w. Make it a single string that depends only on N. Choose w so that it makes your proof easier. You may end up with various cases with different q values that reach a contradiction. You have to show that all possible cases lead to a contradiction.

**Proof of the Pumping Lemma**

Since L is regular it is accepted by some DFSA, M. Let N be the number of states in M. Let w be a string in L of length N or more.

$$\begin{array}{ccccccc}
& & & & N & & \\
& & a & a & a & a & a & a & a & a & b & b & b & b & b & b & b & b & \\
& & x & & y & & & & & & & & & & & & & & & \\
& & x & & y & & & & & & & & & & & & & & & & 
\end{array}$$

Then, in the first N steps of the computation of M on w, M must visit N+1 states. But there are only N different states, so it must have visited the same state more than once. Thus it must have looped at least once. We’ll call the portion of w that corresponds to the loop y. But if it can loop once, it can loop an infinite number of times. Thus:

- M can recognize xy<sup>q</sup>z for all values of q ≥ 0.
- y ≠ ε (since there was a loop of length at least one)
- |xy| ≤ N (since we found y within the first N steps of the computation)

### Another Pumping Example

$L = \{w=a^Jb^K : K > J\}$  (more b's than a's)

Choose  $w = a^N b^{N+1}$

$$\frac{\text{a a a a a a a a a a}}{x} \frac{\text{a a a a a a a a a a}}{y} \frac{\text{b b b b b b b b b b}}{z}$$

We are guaranteed to pump only a's, since  $|xy| \leq N$ . So there exists a number of copies of y that will cause there to be more a's than b's, thus violating the claim that the pumped string is in L.

### A Slightly Different Example of Pumping

$L = \{w=a^Jb^K : J > K\}$  (more a's than b's)

Choose  $w = a^{N+1} b^N$

$$\frac{\text{a a a a a a a a a a}}{x} \frac{\text{a a a a a a a a a a}}{y} \frac{\text{b b b b b b b b b b}}{z}$$

We are guaranteed that y is a string of at least one a, since  $|xy| \leq N$ . But if we pump in a's we get even more a's than b's, resulting in strings that are in L.

What can we do?

### Another Slightly Different Example of Pumping

$L = \{w=a^Jb^K : J \geq K\}$

Choose  $w = a^{N+1} b^N$

$$\frac{\text{a a a a a a a a a a}}{x} \frac{\text{a a a a a a a a a a}}{y} \frac{\text{b b b b b b b b b b}}{z}$$

We are guaranteed that y is a string of at least one a, since  $|xy| \leq N$ . But if we pump in a's we get even more a's than b's, resulting in strings that are in L.

If we pump out, then if y is just a then we still have a string in L.

What can we do?

### Another Pumping Example

$$L = aba^n b^n$$

Choose  $w = aba^N b^N$

$$\begin{array}{cccccccccccccccccccc} a & b & a & a & a & a & a & a & a & a & a & a & b & b & b & b & b & b & b & b & b & b & b \\ \hline x & & y & z \end{array}$$

What are the choices for  $(x, y)$ :

- ( $\epsilon, a$ )
- ( $\epsilon, ab$ )
- ( $\epsilon, aba^+$ )
- ( $a, b$ )
- ( $a, ba^+$ )
- ( $aba^*, a^+$ )

### What if L is Regular?

Given a language L that is regular, pumping will work:

$$L = (ab)^*$$

Choose  $w = (ab)^N$

There must exist an x, y, and z where y is pumpable.

$$\begin{array}{ccccccc} abababab & ababab & ababababab \\ \hline x & & y & & & & z \end{array}$$

Suppose  $y = ababab$

Then, for all  $q \geq 0$ ,  $x y^q z \in L$

Note that this does not prove that L is regular. It just fails to prove that it is not.

### Using Closure Properties

Once we have some languages that we can prove are not regular, such as  $a^n b^n$ , we can use the closure properties of regular languages to show that other languages are also not regular.

Example:  $\Sigma = \{a, b\}$   
 $L = \{w : w \text{ contains an equal number of } a\text{'s and } b\text{'s}\}$   
 $a^*b^*$  is regular. So, if L is regular, then  $L_1 = L \cap a^*b^*$  is regular.

But  $L_1$  is precisely  $a^n b^n$ . So L is not regular.

### Don't Try to Use Closure Backwards

One Closure Theorem:

If  $L_1$  and  $L_2$  are regular, then so is  $L_3 = L_1 \cap L_2$ .

But what if  $L_3$  and  $L_1$  are regular? What can we say about  $L_2$ ?

$$\begin{array}{c} L_3 = L_1 \cap L_2 \\ \uparrow \\ ab = ab \cap a^n b^n \end{array}$$

Example:

### A Harder Example of Pumping

$$\Sigma = \{a\}$$

$$L = \{w = a^K : K \text{ is a prime number}\}$$

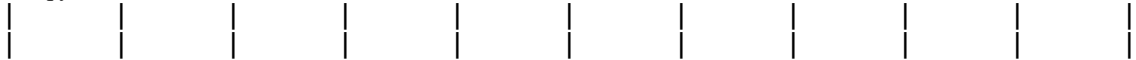
- $|x| + |z|$  is prime.
- $|x| + |y| + |z|$  is prime.
- $|x| + 2|y| + |z|$  is prime.
- $|x| + 3|y| + |z|$  is prime, and so forth.

$$\begin{array}{c} N \\ \hline \text{a a a a a a a a a a a a} \\ \hline \text{x} \quad \text{y} \quad \text{z} \end{array}$$

Distribution of primes:



Distribution of  $|x| + q|y| + |z|$ :



But the Prime Number Theorem tells us that the primes "spread out", i.e., that the number of primes not exceeding  $x$  is asymptotic to  $x/\ln x$ .

Note that when  $q = |x| + |z|$ ,  $|xy^qz| = (|y| + 1)(|x| + |z|)$ , which is composite (non-prime) if both factors are  $> 1$ . If you're careful about how you choose  $N$  in a pumping lemma proof, you can make this true for both factors.

### Automata Theory is Just the Scaffolding

Our results so far give us tools to:

- Show a language is regular by:
  - Showing that it has a finite number of elements,
  - Providing a regular expression that defines it,
  - Constructing a FSA that accepts it, or
  - Exploiting closure properties
- Show a language is not regular by:
  - Using the pumping lemma, or
  - Exploiting closure properties.

But to use these tools effectively, we may also need domain knowledge (e.g., the Prime Number Theorem).

### More Examples

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7\}$$

$$L = \{w = \text{the octal representation of a number that is divisible by 7}\}$$

Example elements of  $L$ :  
 7, 16 (14), 43 (35), 61 (49), 223 (147)

### More Examples

$$\Sigma = \{W, H, Q, E, S, T, B \text{ (measure bar)}\}$$

$$L = \{w = w \text{ represents a song written in } 4/4 \text{ time}\}$$

Example element of  $L$ :  
 WBWBHBBHQBBHQBEEQEEB

### More Examples

$\Sigma = \{0 - 9\}$

$L = \{w \mid w \text{ is a prime Fermat number}\}$

The Fermat numbers are defined by

$$F_n = 2^{2^n} + 1, n = 1, 2, 3, \dots$$

Example elements of L:

$$F_1 = 5, F_2 = 17, F_3 = 257, F_4 = 65,537$$

### Another Example

$\Sigma = \{0 - 9, *, =\}$

$L = \{w = a*b=c \mid a, b, c \in \{0-9\}^+ \text{ and } \text{int}(a) * \text{int}(b) = \text{int}(c)\}$

### The Bottom Line

A language is regular if:

OR

### The Bottom Line (Examples)

- The set of decimal representations for nonnegative integers divisible by 2 or 3
- The social security numbers of living US residents.
- Parity checking
- $a^n b^n$
- $a^j b^k$  where  $k > j$
- $a^k$  where  $k$  is prime
- The set of strings over  $\{a, b\}$  that contain an equal number of a's and b's.
- The octal representations of numbers that are divisible by 7
- The songs in 4/4 time
- The set of prime Fermat numbers

### Decision Procedures

A **decision procedure** is an algorithm that answers a question (usually “yes” or “no”) and terminates. The whole idea of a decision procedure itself raises a new class of questions. In particular, we can now ask,

1. Is there a decision procedure for question X?
2. What is that procedure?
3. How efficient is the best such procedure?

Clearly, if we jump immediately to an answer to question 2, we have our answer to question 1. But sometimes it makes sense to answer question 1 first. For one thing, it tells us whether to bother looking for answers to questions 2 and 3.

Examples of Question 1:

Is there a decision procedure, given a regular expression E and a string S, for determining whether S is in  $L(E)$ ?

Is there a decision procedure, given a Turing machine T and an input string S, for determining whether T halts on S?

## Decision Procedures for Regular Languages

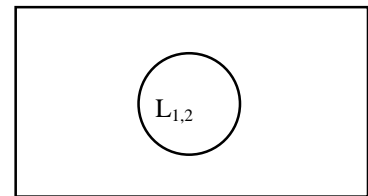
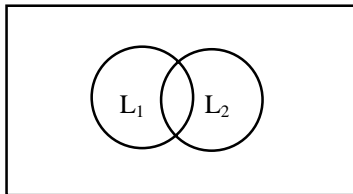
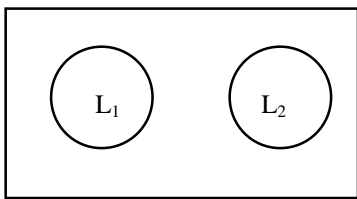
Let  $M$  be a deterministic FSA. There is a decision procedure to determine whether:

- $w \in L(M)$  for some fixed  $w$
- $L(M)$  is empty
- $L(M)$  is finite
- $L(M)$  is infinite

Let  $M_1$  and  $M_2$  be two deterministic FSAs. There is a decision procedure to determine whether  $M_1$  and  $M_2$  are equivalent. Let  $L_1$  and  $L_2$  be the languages accepted by  $M_1$  and  $M_2$ . Then the language

$$\begin{aligned} L &= (L_1 \cap \neg L_2) \cup (\neg L_1 \cap L_2) \\ &= (L_1 - L_2) \cup (L_2 - L_1) \end{aligned}$$

must be regular.  $L$  is empty iff  $L_1 = L_2$ . There is a decision procedure to determine whether  $L$  is empty and thus whether  $L_1 = L_2$  and thus whether  $M_1$  and  $M_2$  are equivalent.



# A Review of Equivalence Relations

Do Homework 7.

## A Review of Equivalence Relations

A relation  $R$  is an equivalence relation if it is: reflexive, symmetric, and transitive.

Example:  $R$  = the reflexive, symmetric, transitive closure of:  
(Bob, Bill), (Bob, Butch), (Butch, Bud),  
(Jim, Joe), (Joe, John), (Joe, Jared),  
(Tim, Tom), (Tom, Tad)

An equivalence relation on a nonempty set  $A$  creates a partition of  $A$ . We write the elements of the partition as  $[a_1], [a_2], \dots$   
Example:

## Another Equivalence Relation

Example:  $R$  = the reflexive, symmetric, transitive closure of:  
(apple, pear), (pear, banana), (pear, peach),  
(peas, mushrooms), (peas, onions), (peas, zucchini)  
(bread, rice), (rice, potatoes), (rice, pasta)

Partition:

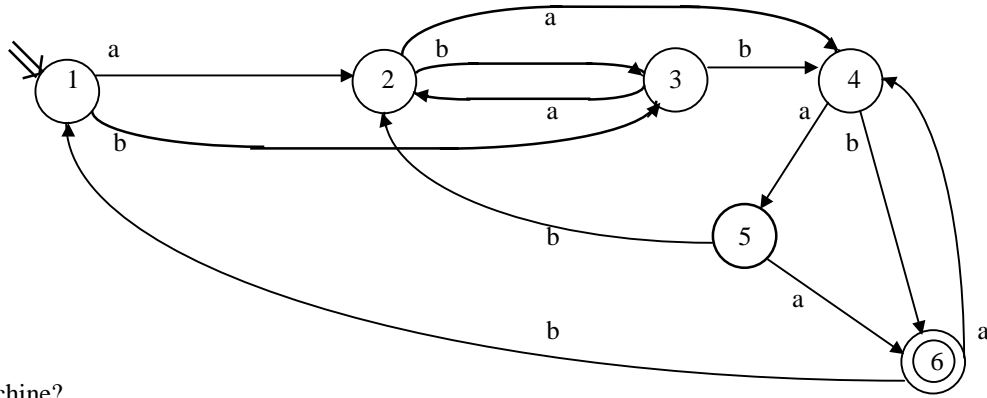


# State Minimization for DFAs

Read K & S 2.7  
Do Homework 10.

Consider:

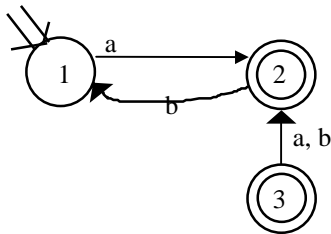
## State Minimization



Is this a minimal machine?

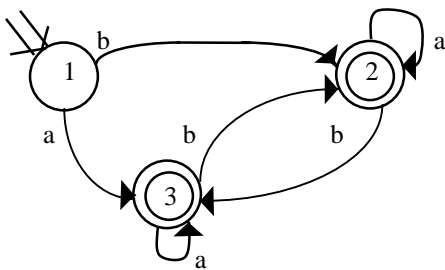
## State Minimization

Step (1): Get rid of unreachable states.



State 3 is unreachable.

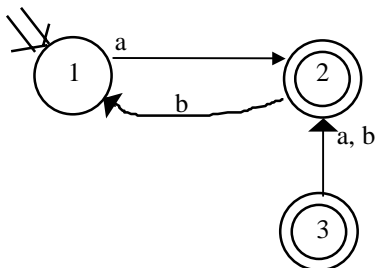
Step (2): Get rid of redundant states.



States 2 and 3 are redundant.

## Getting Rid of Unreachable States

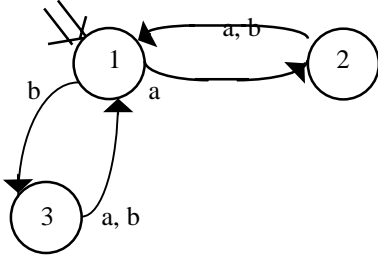
We can't easily find the unreachable states directly. But we can find the reachable ones and determine the unreachable ones from there. An algorithm for finding the reachable states:



## Getting Rid of Redundant States

Intuitively, two states are equivalent to each other (and thus one is redundant) if all strings in  $\Sigma^*$  have the same fate, regardless of which of the two states the machine is in. But how can we tell this?

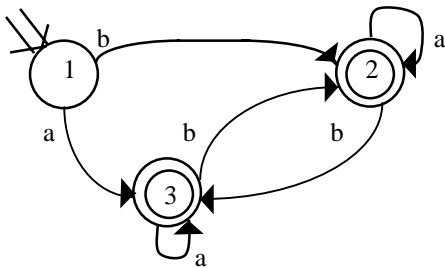
The simple case:



Two states have identical sets of transitions out.

## Getting Rid of Redundant States

The harder case:



The outcomes are the same, even though the states aren't.

## Finding an Algorithm for Minimization

Capture the notion of equivalence classes of strings with respect to a language.

Capture the (weaker) notion of equivalence classes of strings with respect to a language and a particular FSA.

Prove that we can always find a deterministic FSA with a number of states equal to the number of equivalence classes of strings.

Describe an algorithm for finding that deterministic FSA.

## Defining Equivalence for Strings

We want to capture the notion that two strings are equivalent with respect to a language  $L$  if, no matter what is tacked on to them on the right, either they will both be in  $L$  or neither will. Why is this the right notion? Because it corresponds naturally to what the states of a recognizing FSM have to remember.

Example:

(1)	a	b		b	a	b
(2)	b	a		b	a	b

Suppose  $L = \{w \in \{a,b\}^* : |w| \text{ is even}\}$ . Are (1) and (2) equivalent?

Suppose  $L = \{w \in \{a,b\}^* : \text{every } a \text{ is immediately followed by } b\}$ . Are (1) and (2) equivalent?

## Defining Equivalence for Strings

If two strings are equivalent with respect to L, we write  $x \approx_L y$ . Formally,  $x \approx_L y$  if,  $\forall z \in \Sigma^*$ ,  $xz \in L$  iff  $yz \in L$ .

Notice that  $\approx_L$  is an equivalence relation.

**Example:**

$\Sigma = \{a, b\}$

$L = \{w \in \Sigma^* : \text{every } a \text{ is immediately followed by } b \}$

$\epsilon$	aa	bbb
a	bb	baa
b	aba	
	aab	

The equivalence classes of  $\approx_L$ :

$|\approx_L|$  is the number of equivalence classes of  $\approx_L$ .

### Another Example of $\approx_L$

$\Sigma = \{a, b\}$

$L = \{w \in \Sigma^* : |w| \text{ is even}\}$

$\epsilon$	bb	aabb
a	aba	bbaa
b	aab	aabaa
aa	bbb	
	baa	

The equivalence classes of  $\approx_L$ :

### Yet Another Example of $\approx_L$

$\Sigma = \{a, b\}$

$L = aab^*a$

$\epsilon$	ba	aabb
a	bb	aabaa
b	aaa	aabbba
aa	aba	aabbba
ab	aab	
	bab	

The equivalence classes of  $\approx_L$ :

### An Example of $\approx_L$ Where All Elements of L Are Not in the Same Equivalence Class

$\Sigma = \{a, b\}$

$L = \{w \in \{a, b\}^* : \text{no two adjacent characters are the same}\}$

$\epsilon$	bb	aabaa
a	aba	aabbba
b	aab	aabbba
aa	baa	
	aabb	

The equivalence classes of  $\approx_L$ :

### Is $|\approx_L|$ Always Finite?

$\Sigma = \{a, b\}$

$L = a^n b^n$

$\epsilon$

a

b

aa

aba

aaa

aaaa

aaaaa

The equivalence classes of  $\approx_L$ :

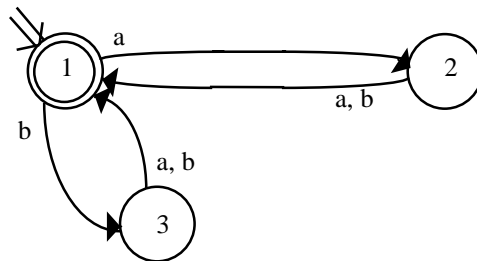
### Bringing FSMs into the Picture

$\approx_L$  is an ideal relation.

What if we now consider what happens to strings when they are being processed by a real FSM?

$\Sigma = \{a, b\}$

$L = \{w \in \Sigma^* : |w| \text{ is even}\}$



Define  $\sim_M$  to relate pairs of strings that drive M from s to the same state.

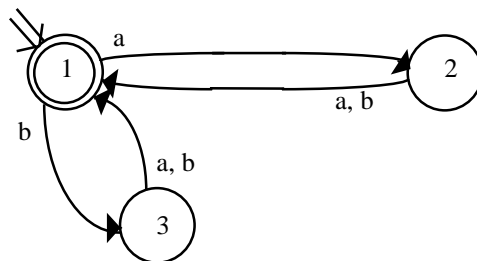
Formally, if M is a deterministic FSM, then  $x \sim_M y$  if there is some state q in M such that  $(s, x) \vdash_M^* (q, \epsilon)$  and  $(s, y) \vdash_M^* (q, \epsilon)$ .

Notice that  $\sim_M$  is an equivalence relation.

### An Example of $\sim_M$

$\Sigma = \{a, b\}$

$L = \{w \in \Sigma^* : |w| \text{ is even}\}$



$\epsilon$

a

b

aa

bb

aba

aab

bbb

baa

aabb

bbaa

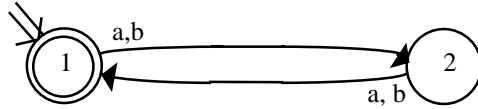
aabaa

The equivalence classes of  $\sim_M$ :

$|\sim_M| =$

### Another Example of $\sim_M$

$$\Sigma = \{a, b\} \quad L = \{w \in \Sigma^* : |w| \text{ is even}\}$$



$\epsilon$	bb	aabb
a	aba	bbaa
b	aab	aabaa
aa	bbb	
	baa	

The equivalence classes of  $\sim_M$ :  $|\sim_M| =$

### The Relationship Between $\approx_L$ and $\sim_M$

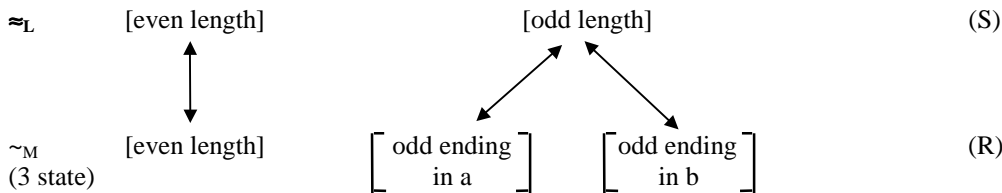
$\approx_L$ :  $[\epsilon, aa, bb, aabb, bbaa] \quad |w| \text{ is even}$   
 $[a, b, aba, aab, bbb, baa, aabaa] \quad |w| \text{ is odd}$

$\sim_M$ , 3 state machine:  
 $q_1: [\epsilon, aa, bb, aabb, bbaa] \quad |w| \text{ is even}$   
 $q_2: [a, aba, baa, aabaa] \quad (ab \cup ba \cup aa \cup bb)^*a$   
 $q_3: [b, aab, bbb] \quad (ab \cup ba \cup aa \cup bb)^*b$

$\sim_M$ , 2 state machine:  
 $q_1: [\epsilon, aa, bb, aabb, bbaa] \quad |w| \text{ is even}$   
 $q_2: [a, b, aba, aab, bbb, baa, aabaa] \quad |w| \text{ is odd}$

$\sim_M$  is a refinement of  $\approx_L$ .

### The Refinement



An equivalence relation R is a refinement of another one S iff

$$xRy \rightarrow xSy$$

In other words, R makes all the same distinctions S does, plus possibly more.

$$|R| \geq |S|$$

### $\sim_M$ is a Refinement of $\approx_L$ .

**Theorem:** For any deterministic finite automaton  $M$  and any strings  $x, y \in \Sigma^*$ , if  $x \sim_M y$ , then  $x \approx_L y$ .

**Proof:** If  $x \sim_M y$ , then  $x$  and  $y$  drive  $M$  to the same state  $q$ . From  $q$ , any continuation string  $w$  will drive  $M$  to some state  $r$ . Thus  $xw$  and  $yw$  both drive  $M$  to  $r$ . Either  $r$  is a final state, in which case they both accept, or it is not, in which case they both reject. But this is exactly the definition of  $\approx_L$ .

**Corollary:**  $|\sim_M| \geq |\approx_L|$ .

### Going the Other Way

When is this true?

If  $x \approx_{L(M)} y$  then  $x \sim_M y$ .

### Finding the Minimal FSM for $L$

What's the smallest number of states we can get away with in a machine to accept  $L$ ?

Example:  $L = \{w \in \Sigma^* : |w| \text{ is even}\}$

The equivalence classes of  $\approx_L$ :

Minimal number of states for  $M(L) =$

This follows directly from the theorem that says that, for any machine  $M$  that accepts  $L$ ,  $|\sim_M|$  must be at least as large as  $|\approx_L|$ .

Can we always find a machine with this minimal number of states?

### The Myhill-Nerode Theorem

**Theorem:** Let  $L$  be a regular language. Then there is a deterministic FSA that accepts  $L$  and that has precisely  $|\approx_L|$  states.

**Proof:** (by construction)

$M =$   
 $K$  states, corresponding to the equivalence classes of  $\approx_L$ .  
 $s = [\epsilon]$ , the equivalence class of  $\epsilon$  under  $\approx_L$ .  
 $F = \{[x] : x \in L\}$   
 $\delta([x], a) = [xa]$

For this construction to prove the theorem, we must show:

1.  $K$  is finite.
2.  $\delta$  is well defined, i.e.,  $\delta([x], a) = [xa]$  is independent of  $x$ .
3.  $L = L(M)$

### The Proof

(1)  $K$  is finite.

Since  $L$  is regular, there must exist a machine  $M$ , with  $|\sim_M|$  finite. We know that

$$|\sim_M| \geq |\approx_L|$$

Thus  $|\approx_L|$  is finite.

(2)  $\delta$  is well defined.

This is assured by the definition of  $\approx_L$ , which groups together precisely those strings that have the same fate with respect to  $L$ .

### The Proof, Continued

(3)  $L = L(M)$

Suppose we knew that  $([x], y) \vdash_M^* ([xy], \epsilon)$ .

Now let  $[x]$  be  $[\epsilon]$  and let  $s$  be a string in  $\Sigma^*$ .

Then

$$([\epsilon], s) \vdash_M^* ([s], \epsilon)$$

$M$  will accept  $s$  if  $[s] \in F$ .

By the definition of  $F$ ,  $[s] \in F$  iff all strings in  $[s]$  are in  $L$ .

So  $M$  accepts precisely the strings in  $L$ .

### The Proof, Continued

**Lemma:**  $([x], y) \vdash_M^* ([xy], \epsilon)$

By induction on  $|y|$ :

Trivial if  $|y| = 0$ .

Suppose true for  $|y| = n$ .

Show true for  $|y| = n+1$

Let  $y = y'a$ , for some character  $a$ . Then,

$$|y'| = n$$

$([x], y'a) \vdash_M^* ([xy'], a)$  (induction hypothesis)

$([xy', ] a) \vdash_M^* ([xy'a], \epsilon)$  (definition of  $\delta$ )

$([x], y'a) \vdash_M^* ([xy'a], \epsilon)$  (trans. of  $\vdash_M^*$ )

$([x], y) \vdash_M^* ([xy], \epsilon)$  (definition of  $y$ )

### Another Version of the Myhill-Nerode Theorem

**Theorem:** A language is regular iff  $|\approx_L|$  is finite.

Example:

Consider:  $L = a^n b^n$   
 $a, aa, aaa, aaaa, aaaaa \dots$

Equivalence classes:

**Proof:**

*Regular  $\rightarrow |\approx_L|$  is finite:* If  $L$  is regular, then there exists an accepting machine  $M$  with a finite number of states  $N$ . We know that  $N \geq |\approx_L|$ . Thus  $|\approx_L|$  is finite.

*$|\approx_L|$  is finite  $\rightarrow$  regular:* If  $|\approx_L|$  is finite, then the standard DFSA  $M_L$  accepts  $L$ . Since  $L$  is accepted by a FSA, it is regular.

## Constructing the Minimal DFA from $\approx_L$

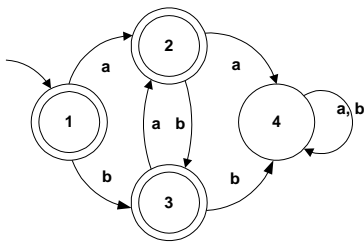
$\Sigma = \{a, b\}$

$L = \{w \in \{a, b\}^* : \text{no two adjacent characters are the same}\}$

The equivalence classes of  $\approx_L$ :

- |                                       |                            |
|---------------------------------------|----------------------------|
| 1: $[\epsilon]$                       | $\epsilon$                 |
| 2: $[a, ba, aba, baba, ababa, \dots]$ | $(b \cup \epsilon)(ab)^*a$ |
| 3: $[b, ab, bab, abab, \dots]$        | $(a \cup \epsilon)(ba)^*b$ |
| 4: $[bb, aa, bba, bbb, \dots]$        | the rest                   |

- Equivalence classes become states
- Start state is  $[\epsilon]$
- Final states are all equivalence classes in  $L$
- $\delta([x], a) = [xa]$



## Using Myhill-Nerode to Prove that $L$ is not Regular

$L = \{a^n : n \text{ is prime}\}$

Consider:

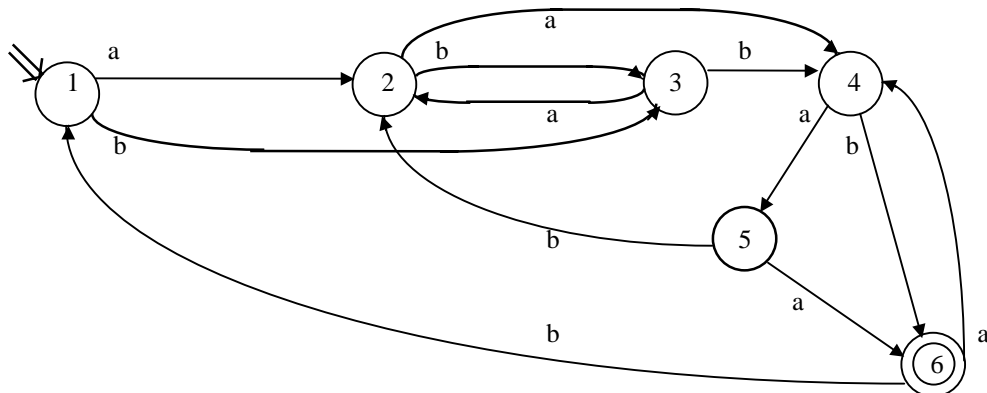
- $\epsilon$
- $a$
- $aa$
- $aaa$
- $aaaa$

Equivalence classes:

### So Where Do We Stand?

1. We know that for any regular language  $L$  there exists a minimal accepting machine  $M_L$ .
  2. We know that  $|K|$  of  $M_L$  equals  $|\approx_L|$ .
  3. We know how to construct  $M_L$  from  $\approx_L$ .
- But is this good enough?

Consider:





## Constructing a Minimal FSA Without Knowing $\approx_L$

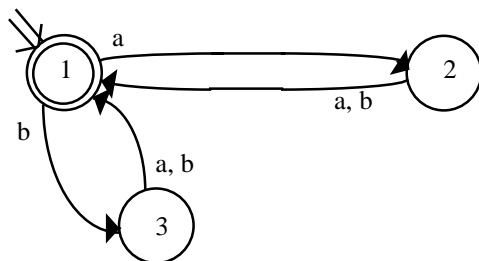
We want to take as input any DFSA  $M'$  that accepts  $L$ , and output a minimal, equivalent DFSA  $M$ .

What we need is a definition for "equivalent", i.e., mergeable states.

Define  $q \equiv p$  iff for all strings  $w \in \Sigma^*$ , either  $w$  drives  $M$  to an accepting state from both  $q$  and  $p$  or it drives  $M$  to a rejecting state from both  $q$  and  $p$ .

Example:

$\Sigma = \{a, b\}$        $L = \{w \in \Sigma^* : |w| \text{ is even}\}$



### Constructing $\equiv$ as the Limit of a Sequence of Approximating Equivalence Relations $\equiv_n$

(Where  $n$  is the length of the input strings that have been considered so far)

We'll consider input strings, starting with  $\epsilon$ , and increasing in length by 1 at each iteration. We'll start by way overgrouping states. Then we'll split them apart as it becomes apparent (with longer and longer strings) that their behavior is not identical.

Initially,  $\equiv_0$  has only two equivalence classes:  $[F]$  and  $[K - F]$ , since on input  $\epsilon$ , there are only two possible outcomes, accept or reject.

Next consider strings of length 1, i.e., each element of  $\Sigma$ . Split any equivalence classes of  $\equiv_0$  that don't behave identically on all inputs. Note that in all cases,  $\equiv_n$  is a refinement of  $\equiv_{n-1}$ .

Continue, until no splitting occurs, computing  $\equiv_n$  from  $\equiv_{n-1}$ .

### Constructing $\equiv$ , Continued

More precisely, for any two states  $p$  and  $q \in K$  and any  $n \geq 1$ ,  $q \equiv_n p$  iff:

1.  $q \equiv_{n-1} p$ , AND
2. for all  $a \in \Sigma$ ,  $\delta(p, a) \equiv_{n-1} \delta(q, a)$

### The Construction Algorithm

The equivalence classes of  $\equiv_0$  are F and K-F.

Repeat for  $n = 1, 2, 3 \dots$

For each equivalence class  $C$  of  $\equiv_{n-1}$  do

For each pair of elements  $p$  and  $q$  in  $C$  do

For each  $a$  in  $\Sigma$  do

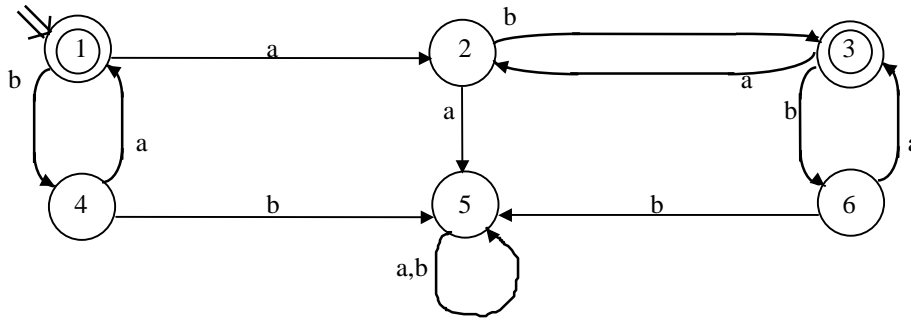
See if  $\delta(p, a) \equiv_{n-1} \delta(q, a)$

If there are any differences in the behavior of  $p$  and  $q$ , then split them and create a new equivalence class.

Until  $\equiv_n = \equiv_{n-1}$ .  $\equiv$  is this answer. Then use these equivalence classes to coalesce states.

### An Example

$\Sigma = \{a, b\}$



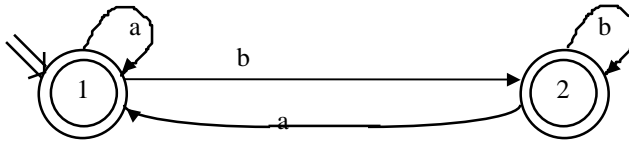
$\equiv_0 =$

$\equiv_1 =$

$\equiv_2 =$

### Another Example

$(a^*b^*)^*$



$\equiv_0 =$

$\equiv_1 =$

Minimal machine:

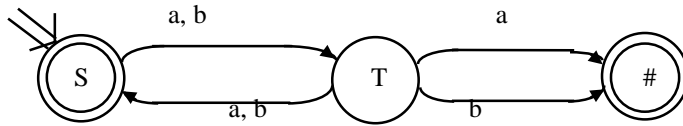
### Another Example

Example:  $L = \{w \in \{a, b\}^* : |w| \text{ is even}\}$

$((aa) \cup (ab) \cup (ba) \cup (bb))^*$

$S \rightarrow \epsilon$   
 $S \rightarrow aT$   
 $S \rightarrow bT$

$T \rightarrow a$   
 $T \rightarrow b$   
 $T \rightarrow aS$   
 $T \rightarrow bS$



Convert to deterministic:

$S = \{s\}$

$\delta =$

### Another Example, Continued

Minimize:



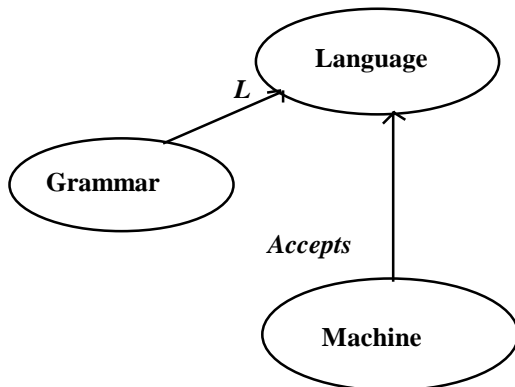
$\equiv_0 =$

$\equiv_1 =$

Minimal machine:

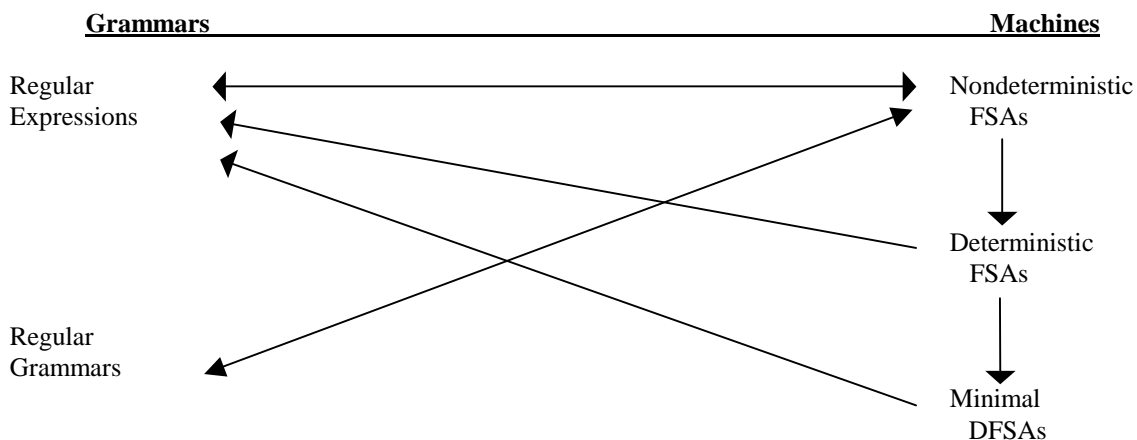
# Summary of Regular Languages and Finite State Machines

## Grammars, Languages, and Machines



## Regular Grammars, Languages, and Machines

Most interesting languages are infinite. So we can't write them down. But we can write down finite grammars and finite machine specifications, and we can define algorithms for mapping between and among them.



## What Does “Finite State” Really Mean?

There are two kinds of finite state problems:

- Those in which:
  - Some history matters.
  - Only a finite amount of history matters. In particular, it's often the case that we don't care what order things occurred in.

Examples:

- Parity
- Money in a vending machine
- Seat belt buzzer
- Those that are characterized by patterns.

Examples:

- Switching circuits:
  - Telephone
  - Railroad
- Traffic lights
- Lexical analysis
- grep

# Context-Free Grammars

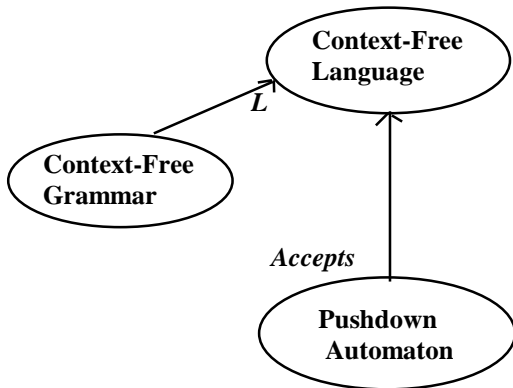
Read K & S 3.1

Read Supplementary Materials: Context-Free Languages and Pushdown Automata: Context-Free Grammars

Read Supplementary Materials: Context-Free Languages and Pushdown Automata: Designing Context-Free Grammars.

Do Homework 11.

## Context-Free Grammars, Languages, and Pushdown Automata



## Grammars Define Languages

Think of grammars as either generators or acceptors.

Example:  $L = \{w \in \{a, b\}^* : |w| \text{ is even}\}$

### Regular Expression

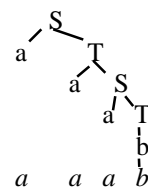
$(aa \cup ab \cup ba \cup bb)^*$

### Regular Grammar

$S \rightarrow \epsilon$   
 $S \rightarrow aT$   
 $S \rightarrow bT$   
 $T \rightarrow a$   
 $T \rightarrow b$   
 $T \rightarrow aS$   
 $T \rightarrow bS$

Derivation  
(Generate)

choose aa  
choose ab  
yields



$a a a b$

Parse (Accept)

use corresponding FSM

## Derivation is Not Necessarily Unique

Example:  $L = \{w \in \{a, b\}^* : \text{there is at least one } a\}$

### Regular Expression

$(a \cup b)^* a (a \cup b)^*$

choose a from  $(a \cup b)$

choose a from  $(a \cup b)$

choose a

choose a

choose a from  $(a \cup b)$

choose a from  $(a \cup b)$

### Regular Grammar

$S \rightarrow a$

$S \rightarrow bS$

$S \rightarrow aS$

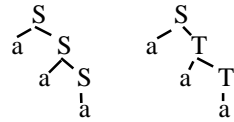
$S \rightarrow aT$

$T \rightarrow a$

$T \rightarrow b$

$T \rightarrow aT$

$T \rightarrow bT$



## More Powerful Grammars

Regular grammars must always produce strings one character at a time, moving left to right.

But sometimes it's more natural to describe generation more flexibly.

Example 1:  $L = ab^*a$

$S \rightarrow aBa$

$B \rightarrow \epsilon$

$B \rightarrow bB$

vs.

$S \rightarrow aB$

$B \rightarrow a$

$B \rightarrow bB$

Example 2:  $L = a^n b^* a^n$

$S \rightarrow B$

$S \rightarrow aSa$

$B \rightarrow \epsilon$

$B \rightarrow bB$

Key distinction: Example 1 has no recursion on the nonregular rule.

## Context-Free Grammars

Remove all restrictions on the form of the right hand sides.

$S \rightarrow abDeFGab$

Keep requirement for single non-terminal on left hand side.

$S \rightarrow$

but not  $ASB \rightarrow$  or  $aSb \rightarrow$  or  $ab \rightarrow$

Examples:

**balanced parentheses**

$S \rightarrow \epsilon$

$S \rightarrow SS$

$S \rightarrow (S)$

**$a^n b^n$**

$S \rightarrow a S b$

$S \rightarrow \epsilon$

## Context-Free Grammars

A context-free grammar  $G$  is a quadruple  $(V, \Sigma, R, S)$ , where:

- $V$  is the rule alphabet, which contains nonterminals (symbols that are used in the grammar but that do not appear in strings in the language) and terminals,
- $\Sigma$  (the set of terminals) is a subset of  $V$ ,
- $R$  (the set of rules) is a finite subset of  $(V - \Sigma) \times V^*$ ,
- $S$  (the start symbol) is an element of  $V - \Sigma$ .

$x \Rightarrow_G y$  is a binary relation where  $x, y \in V^*$  such that  $x = \alpha A \beta$  and  $y = \alpha \chi \beta$  for some rule  $A \rightarrow \chi$  in  $R$ .

Any sequence of the form

$$w_0 \Rightarrow_G w_1 \Rightarrow_G w_2 \Rightarrow_G \dots \Rightarrow_G w_n$$

e.g.,  $(S) \Rightarrow (SS) \Rightarrow ((S)S)$

is called a **derivation in  $G$** . Each  $w_i$  is called a **sentinel form**.

The **language generated by  $G$**  is  $\{w \in \Sigma^* : S \Rightarrow_G^* w\}$

A **language  $L$  is context free** if  $L = L(G)$  for some context-free grammar  $G$ .

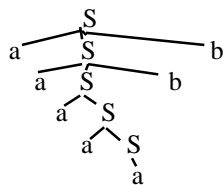
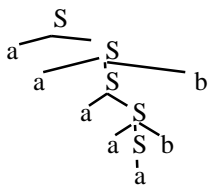
### Example Derivations

$G = (W, \Sigma, R, S)$ , where

$$W = \{S\} \cup \Sigma,$$

$$\Sigma = \{a, b\},$$

$$R = \{ S \rightarrow a, \\ S \rightarrow aS, \\ S \rightarrow aSb \}$$



### Another Example - Unequal a's and b's

$$L = \{a^n b^m : n \neq m\}$$

$G = (W, \Sigma, R, S)$ , where

$$W = \{a, b, S, A, B\},$$

$$\Sigma = \{a, b\},$$

$$R =$$

$$S \rightarrow A$$

$$S \rightarrow B$$

$$A \rightarrow a$$

$$A \rightarrow aA$$

$$A \rightarrow aAb$$

$$B \rightarrow b$$

$$B \rightarrow Bb$$

$$B \rightarrow aBb$$

/\* more a's than b's

/\* more b's than a's

$S \rightarrow NP VP$   
 $NP \rightarrow the NP1 | NP1$   
 $NP1 \rightarrow ADJ NP1 | N$   
 $ADJ \rightarrow big | youngest | oldest$   
 $N \rightarrow boy | boys$   
 $VP \rightarrow V | V NP$   
 $V \rightarrow run | runs$

**English**

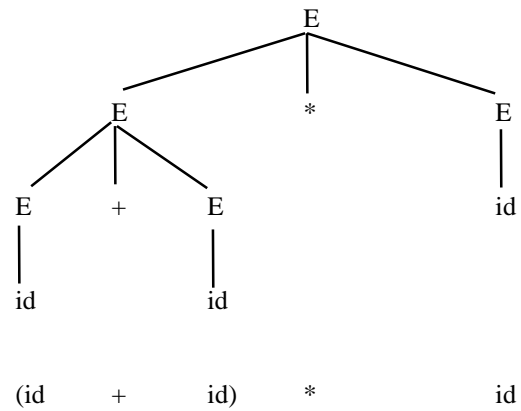
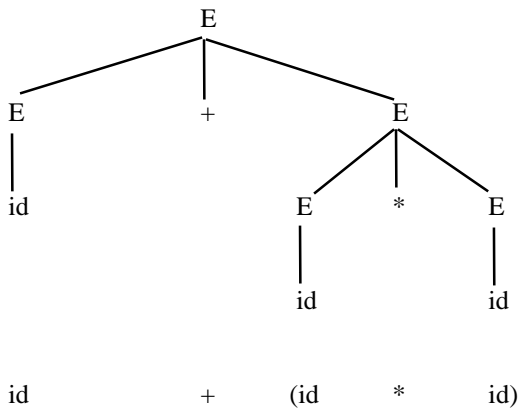
the boys run  
 big boys run  
 the youngest boy runs  
  
 the youngest oldest boy runs  
 the boy run

Who did you say Bill saw coming out of the hotel?

**Arithmetic Expressions**

The Language of Simple Arithmetic Expressions

$G = (V, \Sigma, R, E)$ , where  
 $V = \{+, *, id, T, F, E\}$ ,  
 $\Sigma = \{+, *, id\}$ ,  
 $R = \{ E \rightarrow id$   
 $E \rightarrow E + E$   
 $E \rightarrow E * E \}$



**Arithmetic Expressions -- A Better Way**

The Language of Simple Arithmetic Expressions

$G = (V, \Sigma, R, E)$ , where  
 $V = \{+, *, (, ), id, T, F, E\}$ ,  
 $\Sigma = \{+, *, (, ), id\}$ ,  
 $R = \{ E \rightarrow E + T$   
 $E \rightarrow T$   
 $T \rightarrow T * F$   
 $T \rightarrow F$   
 $F \rightarrow (E)$   
 $F \rightarrow id \}$

Examples:

id + id \* id

id \* id \* id



## BNF

Backus-Naur Form (BNF) is used to define the syntax of programming languages using context-free grammars.

Main idea: give descriptive names to nonterminals and put them in angle brackets.

Example: arithmetic expressions:

$\langle \text{expression} \rangle \rightarrow \langle \text{expression} \rangle + \langle \text{term} \rangle$

$\langle \text{expression} \rangle \rightarrow \langle \text{term} \rangle$

$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle$

$\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle$

$\langle \text{factor} \rangle \rightarrow (\langle \text{expression} \rangle)$

$\langle \text{factor} \rangle \rightarrow \langle \text{id} \rangle$

## The Language of Boolean Logic

$G = (V, \Sigma, R, E)$ , where

$V = \{ \wedge, \vee, \neg, \Rightarrow, (, ), \text{id}, E, E1, E2, E3, E4 \}$ ,

$\Sigma = \{ \wedge, \vee, \neg, \Rightarrow, (, ), \text{id} \}$ ,

$R = \{ E \rightarrow E \Rightarrow E1$

$E \rightarrow E1$

$E1 \rightarrow E1 \vee E2$

$E1 \rightarrow E2$

$E2 \rightarrow E2 \wedge E3$

$E2 \rightarrow E3$

$E3 \rightarrow \neg E4$

$E3 \rightarrow E4$

$E4 \rightarrow (E)$

$E4 \rightarrow \text{id} \}$

## Boolean Logic isn't Regular

Suppose it were regular. Then there is an  $N$  as specified in the pumping theorem.

Let  $w$  be a string of length  $2N + 1 + 2|\text{id}|$  of the form:

$w = \underbrace{(((((((\text{id}))))))}}_N \Rightarrow \text{id}$

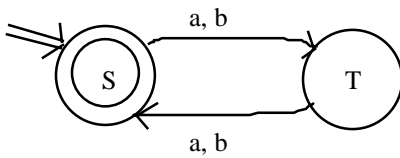
$x \quad y$

$y = \epsilon^k$  for some  $k > 0$  because  $|xy| \leq N$ .

Then the string that is identical to  $w$  except that it has  $k$  additional '('s at the beginning would also be in the language. But it can't be because the parentheses would be mismatched. So the language is not regular.

## All Regular Languages Are Context Free

(1) Every regular language can be described by a regular grammar. We know this because we can derive a regular grammar from any FSM (as well as vice versa). Regular grammars are special cases of context-free grammars.



(2) The context-free languages are precisely the languages accepted by NDPDAs. But every FSM is a PDA that doesn't bother with the stack. So every regular language can be accepted by a NDPDA and is thus context-free.

(3) Context-free languages are closed under union, concatenation, and Kleene \*, and  $\epsilon$  and each single character in  $\Sigma$  are clearly context free.

# Parse Trees

Read K & S 3.2

Read Supplementary Materials: Context-Free Languages and Pushdown Automata: Derivations and Parse Trees.

Do Homework 12.

## Parse Trees

Regular languages:

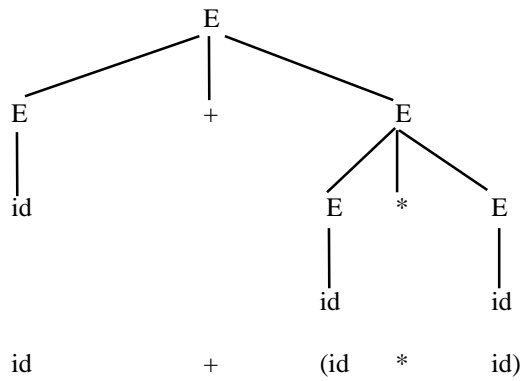
We care about recognizing patterns and taking appropriate actions.

Example: A parity checker

Context free languages:

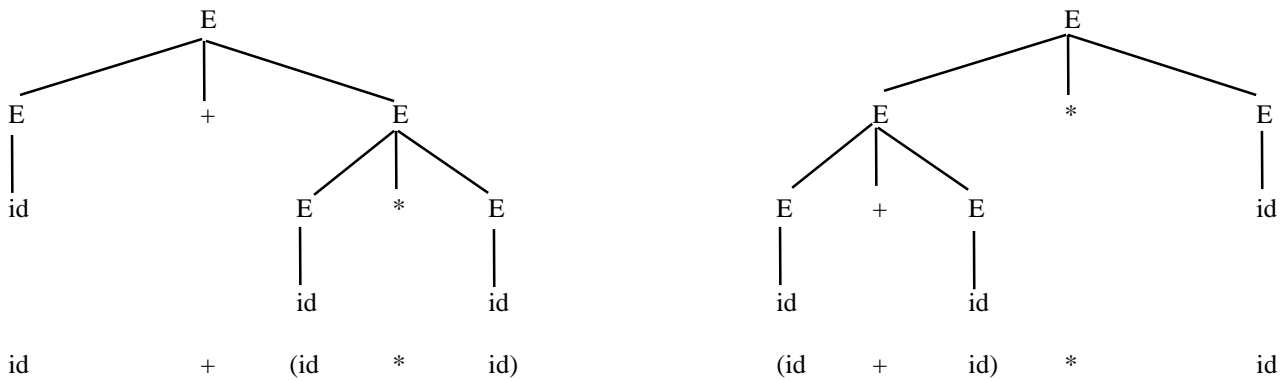
We care about structure.

### Structure

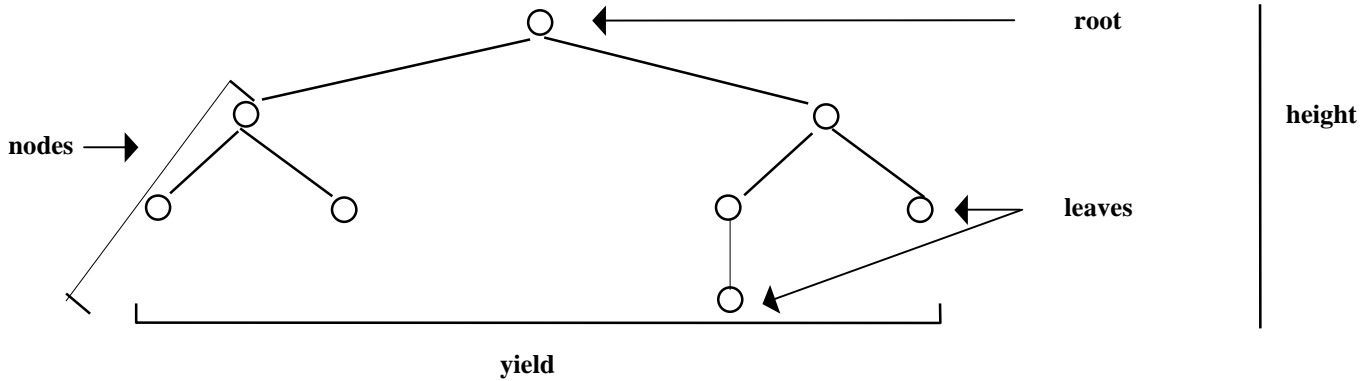


### Parse Trees Capture Essential Structure

- $E \rightarrow id$
- $E \rightarrow E + E$
- $E \rightarrow E * E$



### Parse Trees are Just Trees



Leaves are all labeled with terminals or  $\epsilon$ .  
 Other nodes are labeled with nonterminals.

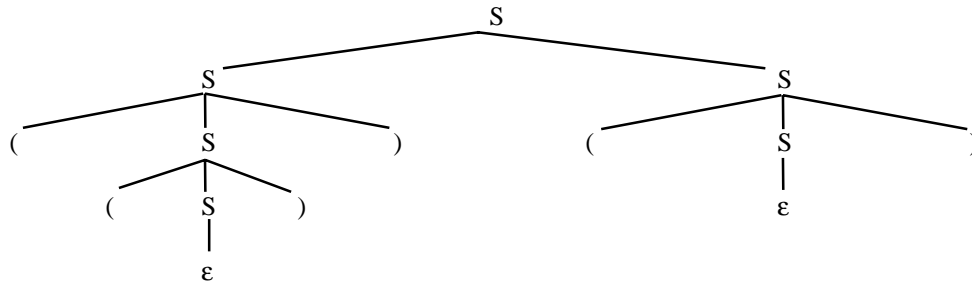
A **path** is a sequence of nodes, starting at the root, ending at a leaf, and following branches in the tree.  
 The length of the yield of any tree  $T$  with height  $H$  and branching factor (fanout)  $B$  is  $\leq$

### Derivations

To capture structure, we must capture the path we took through the grammar. **Derivations** do that.

- $S \rightarrow \epsilon$
- $S \rightarrow SS$
- $S \rightarrow (S)$

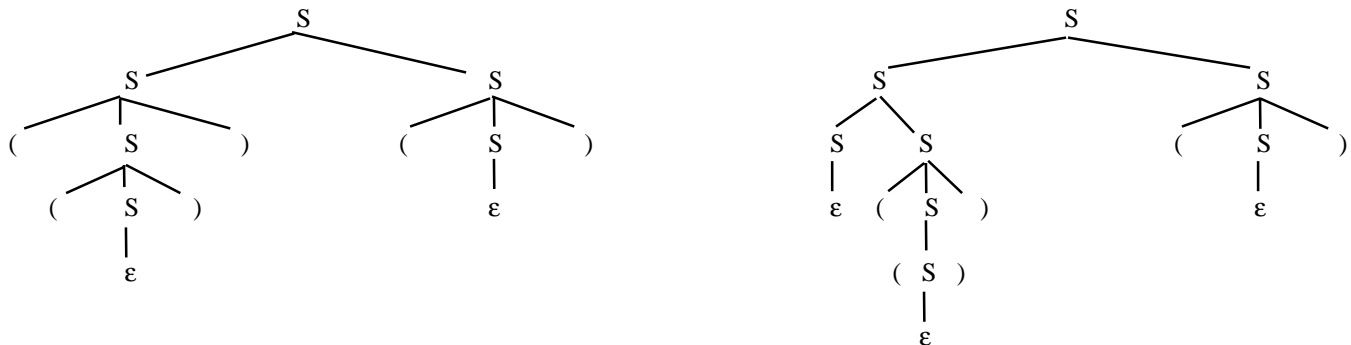
1 2 3 4 5 6  
 $S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (()S) \Rightarrow (()S) \Rightarrow (()()$   
 1 2 3 5 4 6  
 $S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))(S) \Rightarrow (()S) \Rightarrow (()()$



### Alternative Derivations

- $S \rightarrow \epsilon$
- $S \rightarrow SS$
- $S \rightarrow (S)$

$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (()S) \Rightarrow (()S) \Rightarrow (()()$   
 $S \Rightarrow SS \Rightarrow SSS \Rightarrow S(S)S \Rightarrow S((S))S \Rightarrow S(()S) \Rightarrow S(()S) \Rightarrow S(()()) \Rightarrow (()()$



## Ordering Derivations

Consider two derivations:

$$\begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))S \Rightarrow ((S))S \Rightarrow ((S))S \Rightarrow ((S))S \end{matrix}$$

$$\begin{matrix} S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))S \Rightarrow ((S))S \Rightarrow ((S))S \Rightarrow ((S))S \\ 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \end{matrix}$$


We can write these, or any, derivation as

$$D_1 = x_1 \Rightarrow x_2 \Rightarrow x_3 \Rightarrow \dots \Rightarrow x_n$$

$$D_2 = x'_1 \Rightarrow x'_2 \Rightarrow x'_3 \Rightarrow \dots \Rightarrow x'_n$$

We say that  $D_1$  precedes  $D_2$ , written  $D_1 < D_2$ , if:

- $D_1$  and  $D_2$  are the same length  $> 1$ , and
- There is some integer  $k$ ,  $1 < k < n$ , such that:
  - for all  $i \neq k$ ,  $x_i = x'_i$
  - $x_{k-1} = x'_{k-1} = uAvBw : u, v, w \in V^*$ ,  
and  $A, B \in V - \Sigma$
  - $x_k = uyvBw$ , where  $A \rightarrow y \in R$
  - $x'_k = uAvzw$  where  $B \rightarrow z \in R$
  - $x_{k+1} = x'_{k+1} = uyvzw$

## Comparing Several Derivations

Consider three derivations:

$$\begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ (1) S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))S & \Rightarrow ((S))S \Rightarrow ((S))S \Rightarrow ((S))S \\ & & \blacklozenge & & & & \\ (2) S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))S & \Rightarrow ((S))S \Rightarrow ((S))S \Rightarrow ((S))S \\ & & & & \blacklozenge & & \\ (3) S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))S & \Rightarrow ((S))S \Rightarrow ((S))S \Rightarrow ((S))S \end{matrix}$$

$D_1 < D_2$

$D_2 < D_3$

But  $D_1$  does not precede  $D_3$ .

All three seem similar though. We can define similarity:

$D_1$  is **similar** to  $D_2$  iff the pair  $(D_1, D_2)$  is in the reflexive, symmetric, transitive closure of  $<$ .

Note: similar is an equivalence class.

In other words, two derivations are similar if one can be transformed into another by a sequence of switchings in the order of rule applications.

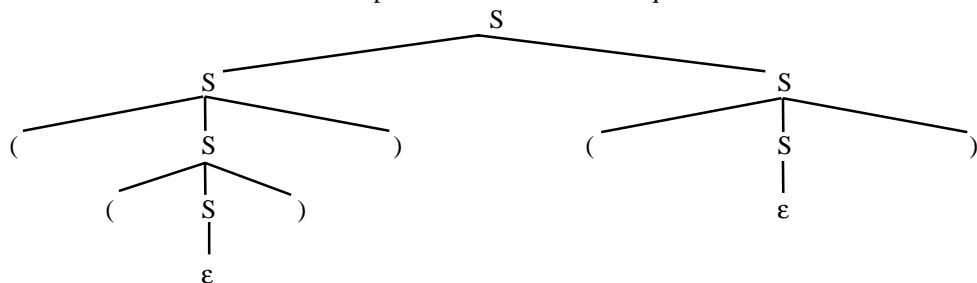
## Parse Trees Capture Similarity

$$\begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ (1) S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))S & \Rightarrow ((S))S \Rightarrow ((S))S \Rightarrow ((S))S \\ & & \blacklozenge & & & & \\ (2) S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))S & \Rightarrow ((S))S \Rightarrow ((S))S \Rightarrow ((S))S \\ & & & & \blacklozenge & & \\ (3) S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))S & \Rightarrow ((S))S \Rightarrow ((S))S \Rightarrow ((S))S \end{matrix}$$

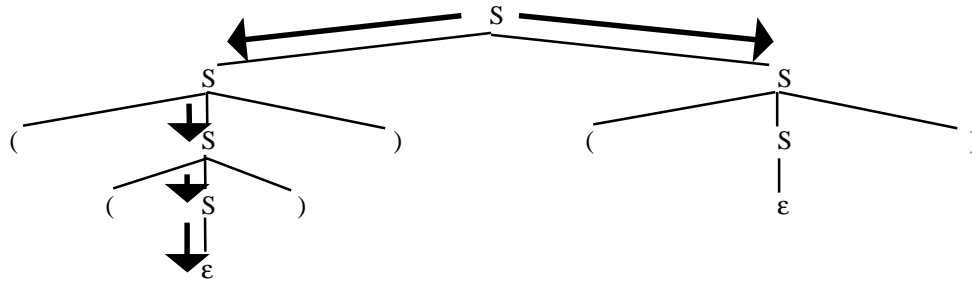
$D_1 < D_2$

$D_2 < D_3$

All three derivations are similar to each other. This parse tree describes this equivalence class of the similarity relation:



### The Maximal Element of $<$



There's one derivation in this equivalence class that precedes all others in the class.

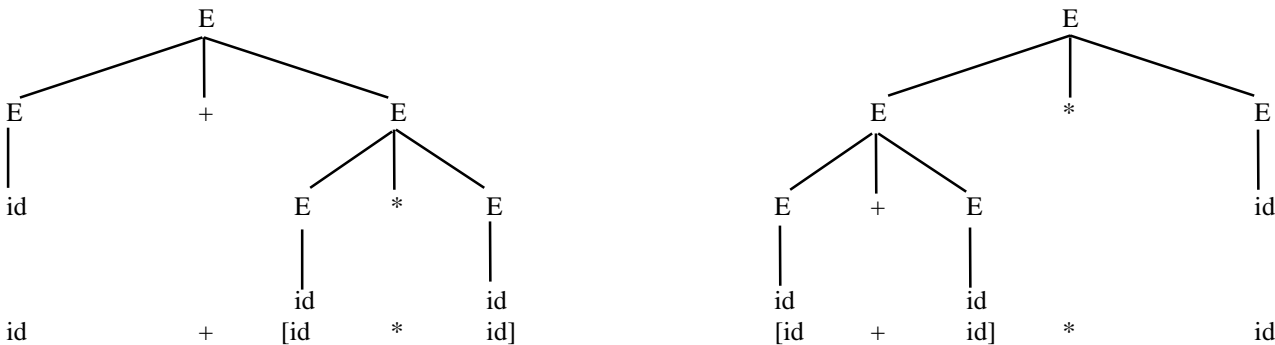
We call this the **leftmost derivation**. There is a corresponding rightmost derivation.

The leftmost (rightmost) derivation can be used to construct the parse tree and the parse tree can be used to construct the leftmost (rightmost) derivation.

### Another Example

- $E \rightarrow id$
- $E \rightarrow E + E$
- $E \rightarrow E * E$

- (1)  $E \Rightarrow E+E \Rightarrow E+E*E \Rightarrow E+E*id \Rightarrow E+id*id \Rightarrow id+id*id$
- (2)  $E \Rightarrow E*E \Rightarrow E*id \Rightarrow E+E*id \Rightarrow E+id*id \Rightarrow id+id*id$



### Ambiguity

A grammar  $G$  for a language  $L$  is **ambiguous** if there exist strings in  $L$  for which  $G$  can generate more than one parse tree (note that we don't care about the number of derivations).

The following grammar for arithmetic expressions is ambiguous:

- $E \rightarrow id$
- $E \rightarrow E + E$
- $E \rightarrow E * E$

Often, when this happens, we can find a different, unambiguous grammar to describe  $L$ .

### Resolving Ambiguity in the Grammar

$G = (V, \Sigma, R, E)$ , where

$V = \{+, *, (, ), id, T, F, E\}$ ,

$\Sigma = \{+, *, (, ), id\}$ ,

$R = \{ E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id \}$

Parse :  $id + id * id$

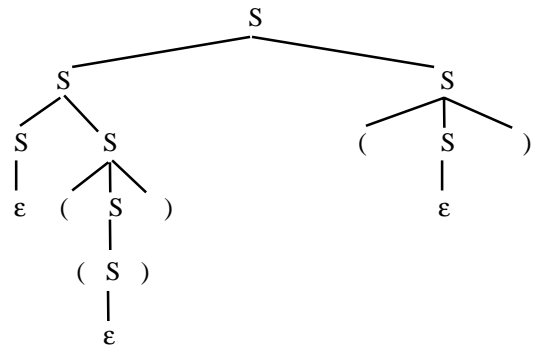
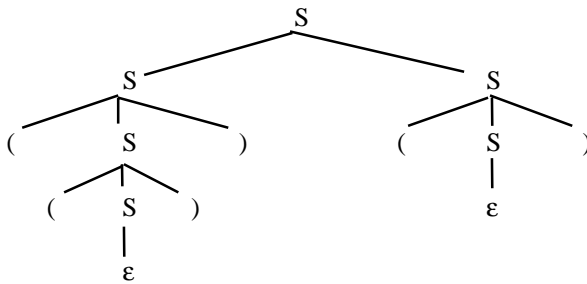
### Another Example

The following grammar for the language of matched parentheses is ambiguous:

$S \rightarrow \epsilon$

$S \rightarrow SS$

$S \rightarrow (S)$



### Resolving the Ambiguity with a Different Grammar

One problem is the  $\epsilon$  production.

A different grammar for the language of balanced parentheses:

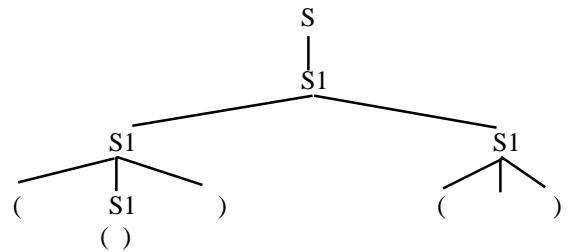
$S \rightarrow \epsilon$

$S \rightarrow S_1$

$S_1 \rightarrow S_1 S_1$

$S_1 \rightarrow (S_1)$

$S_1 \rightarrow ()$



### A General Technique for Eliminating $\epsilon$

If  $G$  is any context-free grammar for a language  $L$  and  $\epsilon \notin L$  then we can construct an alternative grammar  $G'$  for  $L$  by:

1. Find the set  $N$  of nullable variables:

A variable  $V$  is **nullable** if either:

there is a rule

$$(1) V \rightarrow \epsilon$$

or there is a rule

$$(2) V \rightarrow PQR \dots \text{such that } P, Q, R, \dots \text{ are all nullable}$$

So begin with  $N$  containing all the variables that satisfy (1). Evaluate all other variables with respect to (2). Continue until no new variables can be added to  $N$ .

2. For every rule of the form

$$P \rightarrow \alpha Q \beta \text{ for some } Q \text{ in } N, \text{ add a rule}$$

$$P \rightarrow \alpha \beta$$

3. Delete all rules of the form

$$V \rightarrow \epsilon$$

### Sometimes Eliminating Ambiguity Isn't Possible

$$S \rightarrow NP VP$$

The boys hit the ball with the bat.

$$NP \rightarrow \text{the } NP1 \mid NP1 \mid NP2$$

$$NP1 \rightarrow \text{ADJ } NP1 \mid N$$

$$NP2 \rightarrow NP1 PP$$

$$\text{ADJ} \rightarrow \text{big} \mid \text{youngest} \mid \text{oldest}$$

$$N \rightarrow \text{boy} \mid \text{boys} \mid \text{ball} \mid \text{bat} \mid \text{autograph}$$

The boys hit the ball with the autograph.

$$VP \rightarrow V \mid V NP$$

$$VP \rightarrow VP PP$$

$$V \rightarrow \text{hit} \mid \text{hits}$$

$$PP \rightarrow \text{with } NP$$

### Why It's Not Possible

- We could write an unambiguous grammar to describe  $L$  but it wouldn't always get the parses we want. Any grammar that is capable of getting all the parses will be ambiguous because the facts required to choose a derivation cannot be captured in the context-free framework.

Example: Our simple English grammar

[[The boys] [hit [the ball] [with [the bat]]]]

[[The boys] [hit [the ball] [with [the autograph]]]]

- There is no grammar that describes  $L$  that is not ambiguous.

Example:  $L = \{a^n b^n c^m\} \cup \{a^n b^m c^m\}$

$$S \rightarrow S_1 \mid S_2$$

$$S_1 \rightarrow S_1 c \mid A$$

$$A \rightarrow aAb \mid \epsilon$$

$$S_2 \rightarrow aS_2 B$$

$$B \rightarrow bBc \mid \epsilon$$

Now consider the strings  $a^n b^n c^n$

They have two distinct derivations

### Inherent Ambiguity of CFLs

A context free language with the property that all grammars that generate it are ambiguous is **inherently ambiguous**.

$L = \{a^n b^n c^m\} \cup \{a^n b^m c^m\}$  is inherently ambiguous.

Other languages that appear ambiguous given one grammar, turn out not to be inherently ambiguous because we can find an unambiguous grammar.

Examples: Arithmetic Expressions  
Balanced Parentheses

Whenever we design practical languages, it is important that they not be inherently ambiguous.



# Pushdown Automata

Read K & S 3.3.

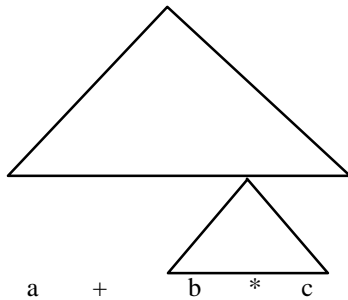
Read Supplementary Materials: Context-Free Languages and Pushdown Automata: Designing Pushdown Automata.

Do Homework 13.

## Recognizing Context-Free Languages

Two notions of recognition:

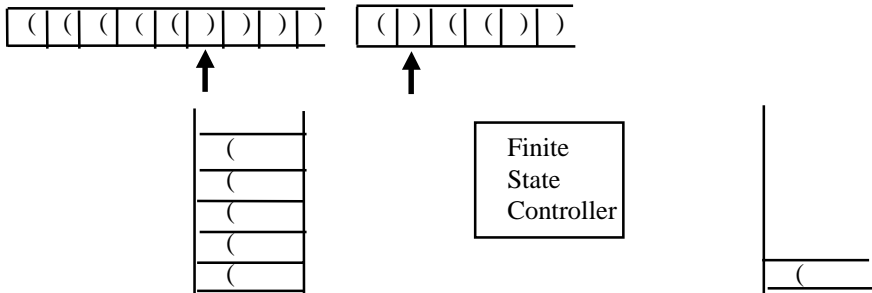
- (1) Say yes or no, just like with FSMs
- (2) Say yes or no, AND  
if yes, describe the structure



### Just Recognizing

We need a device similar to an FSM except that it needs more power.

The insight: Precisely what it needs is a stack, which gives it an unlimited amount of memory with a restricted structure.



### Definition of a Pushdown Automaton

$M = (K, \Sigma, \Gamma, \Delta, s, F)$ , where:

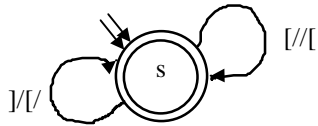
- $K$  is a finite set of states
- $\Sigma$  is the input alphabet
- $\Gamma$  is the stack alphabet
- $s \in K$  is the initial state
- $F \subseteq K$  is the set of final states, and
- $\Delta$  is the transition relation. It is a finite subset of

$$\left( \underbrace{K \times (\Sigma \cup \{\epsilon\}) \times \Gamma^*}_{\text{state input or } \epsilon \text{ string of symbols to pop from top of stack}} \right) \times \left( \underbrace{K \times \Gamma^*}_{\text{state string of symbols to push on top of stack}} \right)$$

$M$  accepts a string  $w$  iff

$$(s, w, \epsilon) \vdash_M^* (p, \epsilon, \epsilon) \quad \text{for some state } p \in F$$

## A PDA for Balanced Brackets



$M = (K, \Sigma, \Gamma, \Delta, s, F)$ , where:

$K = \{s\}$

the states

$\Sigma = \{[, ]\}$

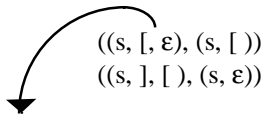
the input alphabet

$\Gamma = \{ \}$

the stack alphabet

$F = \{s\}$

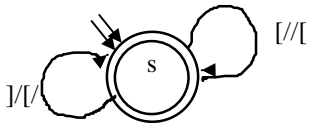
$\Delta$  contains:



Important:

This does not mean that the stack is empty.

### An Example of Accepting



$\Delta$  contains:

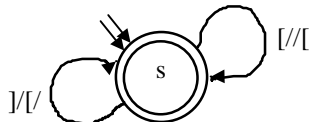
[1]  $((s, [, \epsilon), (s, [ ))$

[2]  $((s, ], [), (s, \epsilon))$

input = [ [ ] [ ] ]

<i>trans</i>	<i>state</i>	<i>unread input</i>	<i>stack</i>
	s	[ [ ] [ ] ]	$\epsilon$
1	s	[ [ ] [ ] ]	[
1	s	[ ] [ ] ]	[ [
1	s	] [ ] ]	[ [ [
2	s	[ ] ] ]	[ [ [
1	s	] ] ] ]	[ [ [ [
2	s	] ] ] ]	[ [ [ [
2	s	] ] ] ]	[ [ [ [
2	s	] ] ] ]	[ [ [ [
2	s	$\epsilon$	$\epsilon$

### An Example of Rejecting



$\Delta$  contains:

[1]  $((s, [, \epsilon), (s, [ ))$

[2]  $((s, ], [), (s, \epsilon))$

input = [ [ ] ] ]

<i>trans</i>	<i>state</i>	<i>unread input</i>	<i>stack</i>
	s	[ [ ] ] ]	$\epsilon$
1	s	[ [ ] ] ]	[
1	s	] ] ] ]	[ [
2	s	] ] ] ]	[
2	s	] ] ] ]	$\epsilon$
none!	s	] ] ] ]	$\epsilon$

We're in s, a final state, but we cannot accept because the input string is not empty. So we reject.

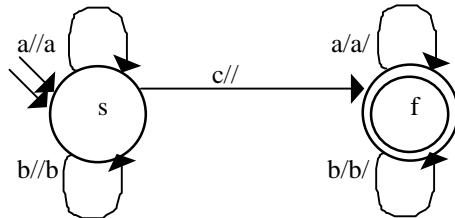
## A PDA for $a^n b^n$

First we notice:

- We'll use the stack to count the a's.
- This time, all strings in L have two regions. So we need two states so that a's can't follow b's. Note the similarity to the regular language  $a^*b^*$ .

## A PDA for $wcw^R$

A PDA to accept strings of the form  $wcw^R$ :



$M = (K, \Sigma, \Gamma, \Delta, s, F)$ , where:

$K = \{s, f\}$

$\Sigma = \{a, b, c\}$

$\Gamma = \{a, b\}$

$F = \{f\}$

$\Delta$  contains:

$((s, a, \epsilon), (s, a))$

$((s, b, \epsilon), (s, b))$

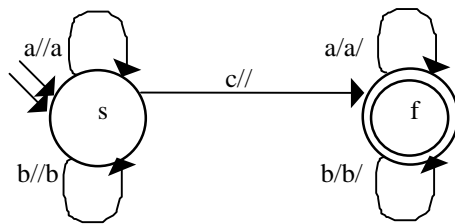
$((s, c, \epsilon), (f, \epsilon))$

$((f, a, a), (f, \epsilon))$

$((f, b, b), (f, \epsilon))$

the states  
the input alphabet  
the stack alphabet  
the final states

### An Example of Accepting



$\Delta$  contains:

[1]  $((s, a, \epsilon), (s, a))$

[2]  $((s, b, \epsilon), (s, b))$

[3]  $((s, c, \epsilon), (f, \epsilon))$

[4]  $((f, a, a), (f, \epsilon))$

[5]  $((f, b, b), (f, \epsilon))$

input = b a c a b

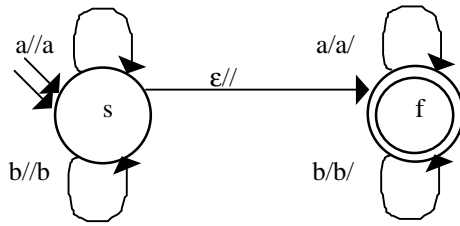
<i>trans</i>	<i>state</i>	<i>unread input</i>	<i>stack</i>
	s	b a c a b	$\epsilon$
2	s	a c a b	b
1	s	c a b	ab
3	f	a b	ab
5	f	b	b
6	f	$\epsilon$	$\epsilon$

## A Nondeterministic PDA

$$L = ww^R$$

- $S \rightarrow \epsilon$
- $S \rightarrow aSa$
- $S \rightarrow bSb$

A PDA to accept strings of the form  $ww^R$ :



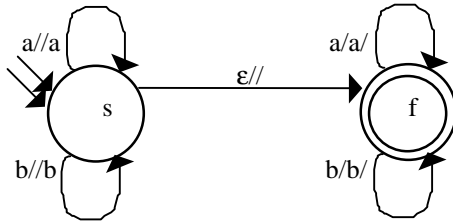
$M = (K, \Sigma, \Gamma, \Delta, s, F)$ , where:

- $K = \{s, f\}$  the states
- $\Sigma = \{a, b, c\}$  the input alphabet
- $\Gamma = \{a, b\}$  the stack alphabet
- $F = \{f\}$  the final states

$\Delta$  contains:

- $((s, a, \epsilon), (s, a))$
- $((s, b, \epsilon), (s, b))$
- $((s, \epsilon, \epsilon), (f, \epsilon))$
- $((f, a, a), (f, \epsilon))$
- $((f, b, b), (f, \epsilon))$

### An Example of Accepting



- |     |                     |  |     |                     |
|-----|---------------------|--|-----|---------------------|
| [1] | ((s, a, ε), (s, a)) |  | [4] | ((f, a, a), (f, ε)) |
| [2] | ((s, b, ε), (s, b)) |  | [5] | ((f, b, b), (f, ε)) |
| [3] | ((s, ε, ε), (f, ε)) |  |     |                     |
- input: a a b b a a

<i>trans</i>	<i>state</i>	<i>unread input</i>	<i>stack</i>
	s	a a b b a a	ε
1	s	a b b a a	a
3	f	a b b a a	a
4	f	b b a a	ε
none			

<i>trans</i>	<i>state</i>	<i>unread input</i>	<i>stack</i>
	s	a a b b a a	ε
1	s	a b b a a	a
1	s	b b a a	aa
2	s	b a a	baa
3	f	b a a	baa
5	f	a a	aa
4	f	a	a
4	f	ε	ε

$$L = \{a^m b^n : m \leq n\}$$

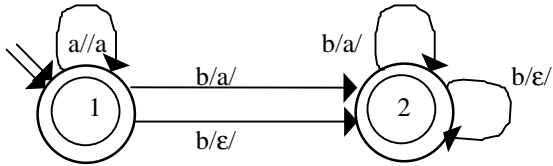
A context-free grammar for L:

$$S \rightarrow \epsilon$$

$$S \rightarrow Sb \quad /* \text{ more b's} */$$

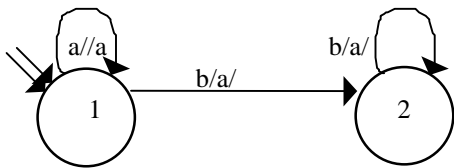
$$S \rightarrow aSb$$

A PDA to accept L:

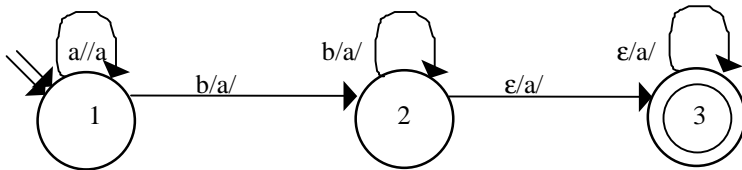


### Accepting Mismatches

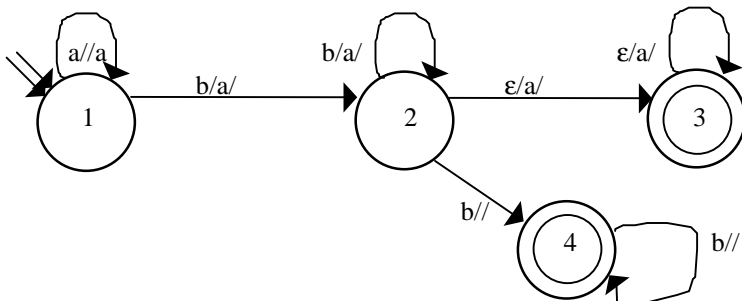
$$L = \{a^m b^n \mid m \neq n; m, n > 0\}$$



- If stack and input are empty, halt and reject.
- If input is empty but stack is not ( $m > n$ ) (accept):

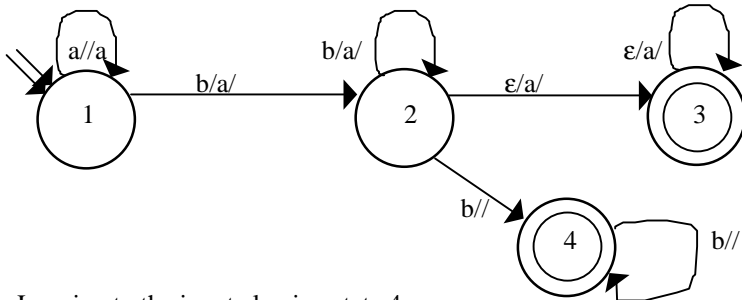


- If stack is empty but input is not ( $m < n$ ) (accept):

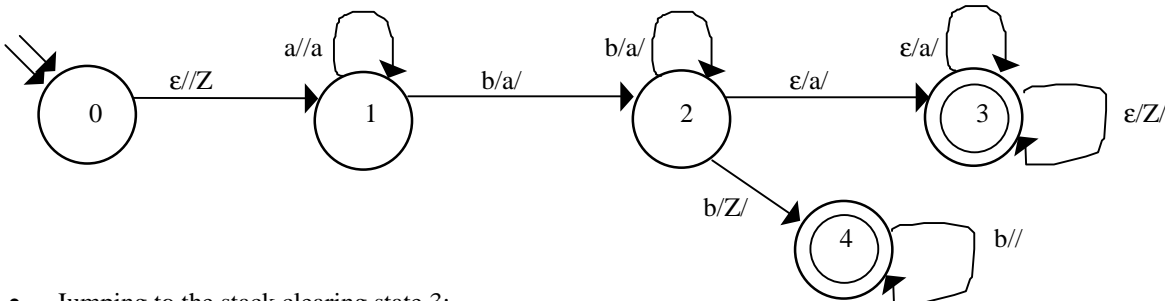


## Eliminating Nondeterminism

A PDA is **deterministic** if, for each input and state, there is at most one possible transition. Determinism implies uniquely defined machine behavior.



- Jumping to the input clearing state 4:  
Need to detect bottom of stack, so push Z onto the stack before we start.

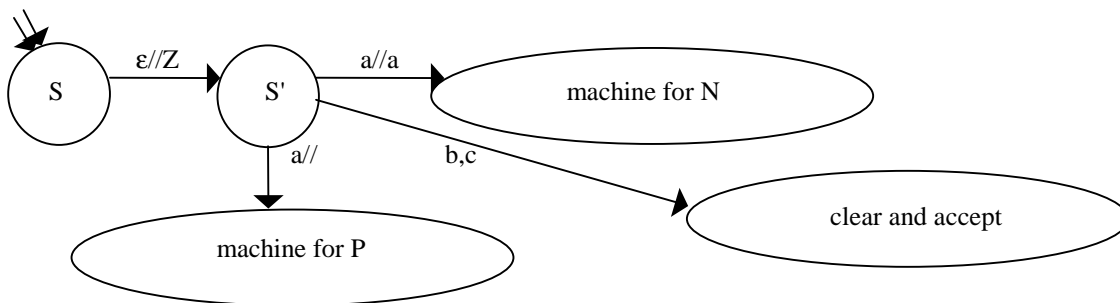


- Jumping to the stack clearing state 3:  
Need to detect end of input. To do that, we actually need to modify the definition of L to add a termination character (e.g., \$)

$$L = \{a^n b^m c^p : n, m, p \geq 0 \text{ and } (n \neq m \text{ or } m \neq p)\}$$

S → NC	/* n ≠ m, then arbitrary c's	C → ε   cC	/* add any number of c's
S → QP	/* arbitrary a's, then p ≠ m	P → B'	/* more b's than c's
N → A	/* more a's than b's	P → C'	/* more c's than b's
N → B	/* more b's than a's	B' → b	
A → a		B' → bB'	
A → aA		B' → bB'c	
A → aAb		C' → c   C'c	
B → b		C' → C'c	
B → Bb		C' → bC'c	
B → aBb		Q → ε   aQ	/* prefix with any number of a's

$$L = \{a^n b^m c^p : n, m, p \geq 0 \text{ and } (n \neq m \text{ or } m \neq p)\}$$



### Another Deterministic CFL

$$L = \{a^n b^n\} \cup \{b^n a^n\}$$

A CFG for L:

- $S \rightarrow A$
- $S \rightarrow B$
- $A \rightarrow \epsilon$
- $A \rightarrow aAb$
- $B \rightarrow \epsilon$
- $B \rightarrow bBa$

A NDPDA for L:

A DPDA for L:

### More on PDAs

What about a PDA to accept strings of the form  $ww$ ?

### Every FSM is (Trivially) a PDA

Given an FSM  $M = (K, \Sigma, \Delta, s, F)$

and elements of  $\Delta$  of the form

$$\left( \begin{array}{ccc} p, & i, & q \\ \text{old state,} & \text{input,} & \text{new state} \end{array} \right)$$

We construct a PDA  $M' = (K, \Sigma, \Gamma, \Delta, s, F)$

where  $\Gamma = \emptyset$  /\* stack alphabet

and

each transition  $(p, i, q)$  becomes

$$\left( \left( \begin{array}{ccc} p, & i, & \epsilon \\ \text{old state,} & \text{input,} & \text{don't look at stack} \end{array} \right), \left( \begin{array}{cc} q, & \epsilon \\ \text{new state} & \text{don't push on stack} \end{array} \right) \right)$$

In other words, we just don't use the stack.

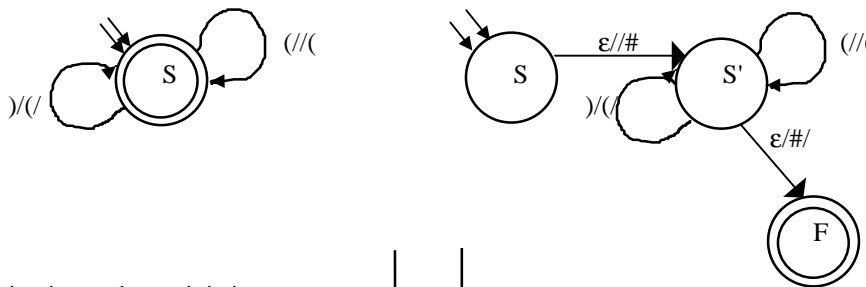
### Alternative (but Equivalent) Definitions of a NDPDA

*Example:* Accept by final state at end of string (i.e., we don't care about the stack being empty)

We can easily convert from one of our machines to one of these:

1. Add a new state at the beginning that pushes # onto the stack.
2. Add a new final state and a transition to it that can be taken if the input string is empty and the top of the stack is #.

Converting the balanced parentheses machine:

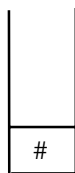


The new machine is nondeterministic:

$$\left( \right) \left( \right)$$

↑

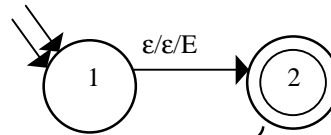
The stack will be:



## What About PDA's for Interesting Languages?

$E \rightarrow E + T$   
 $E \rightarrow T$   
 $T \rightarrow T * F$   
 $T \rightarrow F$   
 $F \rightarrow (E)$   
 $F \rightarrow \text{id}$

Arithmetic Expressions



- (1) (2, ε, E), (2, E+T)
- (2) (2, ε, E), (2, T)
- (3) (2, ε, T), (2, T\*F)
- (4) (2, ε, T), (2, F)
- (5) (2, ε, F), (2, (E) )
- (6) (2, ε, F), (2, id)
- (7) (2, id, id), (2, ε)
- (8) (2, (, ( ), (2, ε)
- (9) (2, ), ) ), (2, ε)
- (10) (2, +, +), (2, ε)
- (11) (2, \*, \*), (2, ε)

*Example:*

a + b \* c

But what we really want to do with languages like this is to extract structure.

## Comparing Regular and Context-Free Languages

### Regular Languages

- regular expressions
- or -
- regular grammars
- recognize
- = DFSAs

### Context-Free Languages

- context-free grammars
- parse
- = NDPDAs



# Pushdown Automata and Context-Free Grammars

Read K & S 3.4.

Read Supplementary Materials: Context-Free Languages and Pushdown Automata: Context-Free Languages and PDAs.  
Do Homework 14.

## PDAs and Context-Free Grammars

**Theorem:** The class of languages accepted by PDAs is exactly the class of context-free languages.

Recall: context-free languages are languages that can be defined with context-free grammars.

**Restate theorem:** Can describe with context-free grammar  $\Leftrightarrow$  Can accept by PDA

### Going One Way

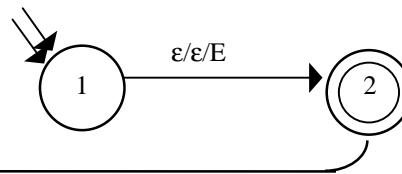
**Lemma:** Each context-free language is accepted by some PDA.

Proof (by construction by “top-down parse” conversion algorithm):

The idea: Let the stack do the work.

Example: Arithmetic expressions

$E \rightarrow E + T$   
 $E \rightarrow T$   
 $T \rightarrow T * F$   
 $T \rightarrow F$   
 $F \rightarrow (E)$   
 $F \rightarrow id$



- |                          |                         |
|--------------------------|-------------------------|
| (1) (2, ε, E), (2, E+T)  | (7) (2, id, id), (2, ε) |
| (2) (2, ε, E), (2, T)    | (8) (2, (, ( ), (2, ε)  |
| (3) (2, ε, T), (2, T*F)  | (9) (2, ), ) ), (2, ε)  |
| (4) (2, ε, T), (2, F)    | (10) (2, +, +), (2, ε)  |
| (5) (2, ε, F), (2, (E) ) | (11) (2, *, *), (2, ε)  |
| (6) (2, ε, F), (2, id)   |                         |

### The Top-down Parse Conversion Algorithm

Given  $G = (V, \Sigma, R, S)$

Construct  $M$  such that  $L(M) = L(G)$

$M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\})$ , where  $\Delta$  contains:

- (1)  $((p, \epsilon, \epsilon), (q, S))$   
push the start symbol on the stack
- (2)  $((q, \epsilon, A), (q, x))$  for each rule  $A \rightarrow x$  in  $R$   
replace left hand side with right hand side
- (3)  $((q, a, a), (q, \epsilon))$  for each  $a \in \Sigma$   
read an input character and pop it from the stack

The resulting machine can execute a leftmost derivation of an input string in a top-down fashion.

### Example of the Algorithm

$$L = \{a^n b^m a^n\}$$

(1)	$S \rightarrow \epsilon$	0	$(p, \epsilon, \epsilon), (q, S)$
(2)	$S \rightarrow B$	1	$(q, \epsilon, S), (q, \epsilon)$
(3)	$S \rightarrow aSa$	2	$(q, \epsilon, S), (q, B)$
(4)	$B \rightarrow \epsilon$	3	$(q, \epsilon, S), (q, aSa)$
(5)	$B \rightarrow bB$	4	$(q, \epsilon, B), (q, \epsilon)$
		5	$(q, \epsilon, B), (q, bB)$
		6	$(q, a, a), (q, \epsilon)$
		7	$(q, b, b), (q, \epsilon)$

input = a a b b a a

<i>trans</i>	<i>state</i>	<i>unread input</i>	<i>stack</i>
	p	a a b b a a	$\epsilon$
0	q	a a b b a a	S
3	q	a a b b a a	aSa
6	q	a b b a a	Sa
3	q	a b b a a	aSaa
6	q	b b a a	Saa
2	q	b b a a	Baa
5	q	b b a a	bBaa
7	q	b a a	Baa
5	q	b a a	bBaa
7	q	a a	Baa
4	q	a a	aa
6	q	a	a
6	q	$\epsilon$	$\epsilon$

### Another Example

$$L = \{a^n b^m c^p d^q : m + n = p + q\}$$

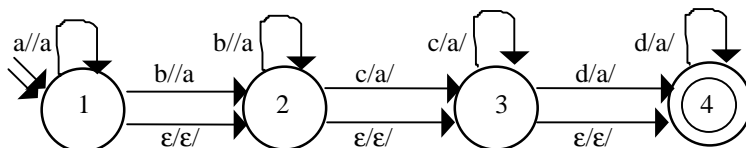
(1)	$S \rightarrow aSd$	0	$(p, \epsilon, \epsilon), (q, S)$
(2)	$S \rightarrow T$	1	$(q, \epsilon, S), (q, aSd)$
(3)	$S \rightarrow U$	2	$(q, \epsilon, S), (q, T)$
(4)	$T \rightarrow aTc$	3	$(q, \epsilon, S), (q, U)$
(5)	$T \rightarrow V$	4	$(q, \epsilon, T), (q, aTc)$
(6)	$U \rightarrow bUd$	5	$(q, \epsilon, T), (q, V)$
(7)	$U \rightarrow V$	6	$(q, \epsilon, U), (q, bUd)$
(8)	$V \rightarrow bVc$	7	$(q, \epsilon, U), (q, V)$
(9)	$V \rightarrow \epsilon$	8	$(q, \epsilon, V), (q, bVc)$
		9	$(q, \epsilon, V), (q, \epsilon)$
		10	$(q, a, a), (q, \epsilon)$
		11	$(q, b, b), (q, \epsilon)$
		12	$(q, c, c), (q, \epsilon)$
		13	$(q, d, d), (q, \epsilon)$

input = a a b c d d

### The Other Way—Build a PDA Directly

$$L = \{a^n b^m c^p d^q : m + n = p + q\}$$

(1)	$S \rightarrow aSd$	(6)	$U \rightarrow bUd$
(2)	$S \rightarrow T$	(7)	$U \rightarrow V$
(3)	$S \rightarrow U$	(8)	$V \rightarrow bVc$
(4)	$T \rightarrow aTc$	(9)	$V \rightarrow \epsilon$
(5)	$T \rightarrow V$		



input = a a b c d d

### Notice Nondeterminism

Machines constructed with the algorithm are often nondeterministic, even when they needn't be. This happens even with trivial languages.

Example:  $L = a^n b^n$

A grammar for L is:

- [1]  $S \rightarrow aSb$
- [2]  $S \rightarrow \epsilon$

A machine M for L is:

- (0)  $((p, \epsilon, \epsilon), (q, S))$
- (1)  $((q, \epsilon, S), (q, aSb))$
- (2)  $((q, \epsilon, S), (q, \epsilon))$
- (3)  $((q, a, a), (q, \epsilon))$
- (4)  $((q, b, b), (q, \epsilon))$

But transitions 1 and 2 make M nondeterministic.

A **nondeterministic transition group** is a set of two or more transitions out of the same state that can fire on the same configuration. A **PDA is nondeterministic** if it has any nondeterministic transition groups.

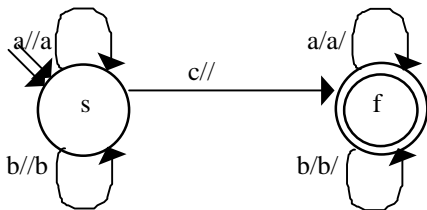
A directly constructed machine for L:

### Going The Other Way

**Lemma:** If a language is accepted by a pushdown automaton, it is a context-free language (i.e., it can be described by a context-free grammar).

Proof (by construction)

Example:  $L = \{wcw^R : w \in \{a, b\}^*\}$



$\Delta$  contains:

- $((s, a, \epsilon), (s, a))$
- $((s, b, \epsilon), (s, b))$
- $((s, c, \epsilon), (f, \epsilon))$
- $((f, a, a), (f, \epsilon))$
- $((f, b, b), (f, \epsilon))$

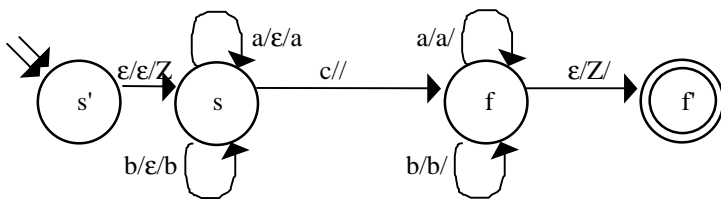
$M = (\{s, f\}, \{a, b, c\}, \{a, b\}, \Delta, s, \{f\})$ , where:

### First Step: Make M Simple

A PDA M is simple iff:

1. there are no transitions into the start state, and
2. whenever  $((q, x, \beta), (p, \gamma))$  is a transition of M and q is not the start state, then  $\beta \in \Gamma$ , and  $|\gamma| \leq 2$ .

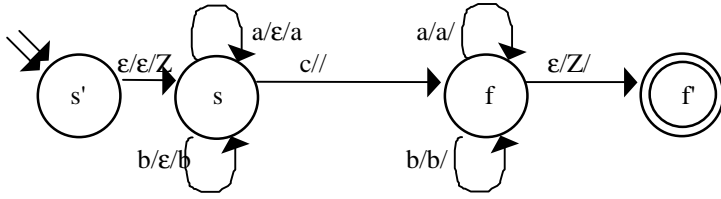
Step 1: Add s' and f':



Step 2:

- (1) Assure that  $|\beta| \leq 1$ .
- (2) Assure that  $|\gamma| \leq 2$ .
- (3) Assure that  $|\beta| = 1$ .

## Making M Simple



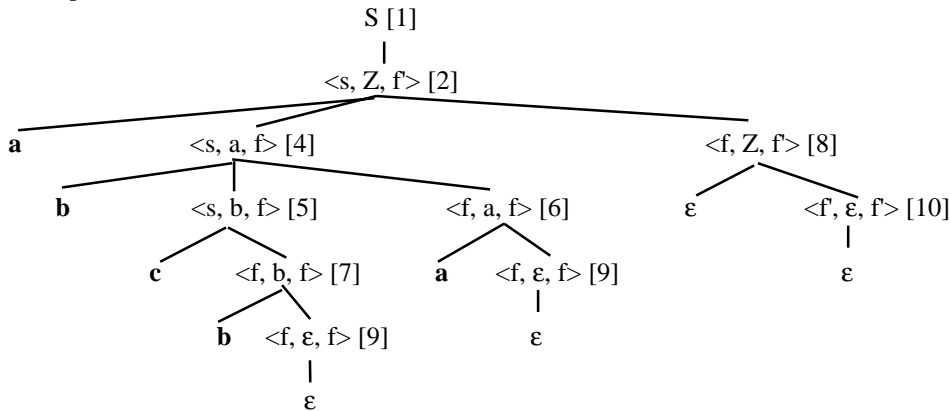
$M = (\{s, f, s', f'\}, \{a, b, c\}, \{a, b, Z\}, \Delta, s', \{f'\}), \Delta =$

	((s', $\epsilon$ , $\epsilon$ ), (s, Z))
((s, a, $\epsilon$ ), (s, a))	((s, a, Z), (s, aZ))
	((s, a, a), (s, aa))
	((s, a, b), (s, ab))
((s, b, $\epsilon$ ), (s, b))	((s, b, Z), (s, bZ))
	((s, b, a), (s, ba))
	((s, b, b), (s, bb))
((s, c, $\epsilon$ ), (f, $\epsilon$ ))	((s, c, Z), (f, Z))
	((s, c, a), (f, a))
	((s, c, b), (f, b))
((f, a, a), (f, $\epsilon$ ))	((f, a, a), (f, $\epsilon$ ))
((f, b, b), (f, $\epsilon$ ))	((f, b, b), (f, $\epsilon$ ))
	((f, $\epsilon$ , Z), (f, $\epsilon$ ))

### Second Step - Creating the Productions

The basic idea -- simulate a leftmost derivation of M on any input string.

Example:            abcba



If the nonterminal  $\langle s_1, X, s_2 \rangle \Rightarrow^* w$ , then the PDA starts in state  $s_1$  with (at least)  $X$  on the stack and after consuming  $w$  and popping the  $X$  off the stack, it ends up in state  $s_2$ .

Start with the rule:

$S \rightarrow \langle s, Z, f' \rangle$  where  $s$  is the start state,  $f'$  is the (introduced) final state and  $Z$  is the stack bottom symbol.

Transitions  $((s_1, a, X), (s_2, YX))$  become a set of rules:

$\langle s_1, X, q \rangle \rightarrow a \langle s_2, Y, r \rangle \langle r, X, q \rangle$  for  $a \in \Sigma \cup \{\epsilon\}, \forall q, r \in K$

Transitions  $((s_1, a, X), (s_2, Y))$  becomes a set of rules:

$\langle s_1, X, q \rangle \rightarrow a \langle s_2, Y, q \rangle$  for  $a \in \Sigma \cup \{\epsilon\}, \forall q \in K$

Transitions  $((s_1, a, X), (s_2, \epsilon))$  become a rule:

$\langle s_1, X, s_2 \rangle \rightarrow a$  for  $a \in \Sigma \cup \{\epsilon\}$

### Creating Productions from Transitions

	$S \rightarrow \langle s, Z, f \rangle$	[1]
$((s', \epsilon, \epsilon), (s, Z))$		
$((s, a, Z), (s, aZ))$	$\langle s, Z, f \rangle \rightarrow a \langle s, a, f \rangle \langle f, Z, f \rangle$	[2]
	$\langle s, Z, s \rangle \rightarrow a \langle s, a, f \rangle \langle f, Z, s \rangle$	[x]
	$\langle s, Z, f \rangle \rightarrow a \langle s, a, s \rangle \langle s, Z, f \rangle$	[x]
	$\langle s, Z, s \rangle \rightarrow a \langle s, a, s \rangle \langle s, Z, f \rangle$	[x]
	$\langle s, Z, s' \rangle \rightarrow a \langle s, a, f \rangle \langle f, Z, s' \rangle$	[x]
$((s, a, a), (s, aa))$	$\langle s, a, f \rangle \rightarrow a \langle s, a, f \rangle \langle f, a, f \rangle$	[3]
$((s, a, b), (s, ab))$	...	
$((s, b, Z), (s, bZ))$	...	
$((s, b, a), (s, ba))$	$\langle s, a, f \rangle \rightarrow b \langle s, b, f \rangle \langle f, a, f \rangle$	[4]
$((s, b, b), (s, bb))$	...	
$((s, c, Z), (f, Z))$	...	
$((s, c, a), (f, a))$	$\langle s, a, f \rangle \rightarrow c \langle f, a, f \rangle$	
$((s, c, b), (f, b))$	$\langle s, b, f \rangle \rightarrow c \langle f, b, f \rangle$	[5]
$((f, a, a), (f, \epsilon))$	$\langle f, a, f \rangle \rightarrow a \langle f, \epsilon, f \rangle$	[6]
$((f, b, b), (f, \epsilon))$	$\langle f, b, f \rangle \rightarrow b \langle f, \epsilon, f \rangle$	[7]
$((f, \epsilon, Z), (f', \epsilon))$	$\langle f, Z, f' \rangle \rightarrow \epsilon \langle f', \epsilon, f' \rangle$	[8]
	$\langle f, \epsilon, f \rangle \rightarrow \epsilon$	[9]
	$\langle f' \epsilon, f' \rangle \rightarrow \epsilon$	[10]

### Comparing Regular and Context-Free Languages

#### Regular Languages

- regular exprs.
  - or
- regular grammars
- recognize
- = DFSAs

#### Context-Free Languages

- context-free grammars
- parse
- = NDPDAs

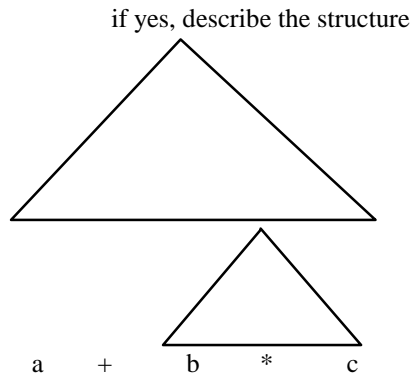
# Grammars and Normal Forms

Read K & S 3.7.

## Recognizing Context-Free Languages

Two notions of recognition:

- (1) Say yes or no, just like with FSMs
- (2) Say yes or no, AND



Now it's time to worry about extracting structure (and **doing so efficiently**).

## Optimizing Context-Free Languages

### For regular languages:

Computation = operation of FSMs. So,

Optimization = Operations on FSMs:

**Conversion to deterministic FSMs**

**Minimization of FSMs**

### For context-free languages:

Computation = operation of parsers. So,

Optimization = **Operations on languages**

**Operations on grammars**

**Parser design**

## Before We Start: Operations on Grammars

There are lots of ways to transform grammars so that they are more useful for a particular purpose.

the basic idea:

1. Apply transformation 1 to G to get rid of undesirable property 1. Show that the language generated by G is unchanged.
2. Apply transformation 2 to G to get rid of undesirable property 2. Show that the language generated by G is unchanged AND that undesirable property 1 has not been reintroduced.
3. Continue until the grammar is in the desired form.

Examples:

- Getting rid of  $\epsilon$  rules (nullable rules)
- Getting rid of sets of rules with a common initial terminal, e.g.,
  - $A \rightarrow aB, A \rightarrow aC$  become  $A \rightarrow aD, D \rightarrow B | C$
- Conversion to normal forms

## Normal Forms

If you want to design algorithms, it is often useful to have a limited number of input forms that you have to deal with.

Normal forms are designed to do just that. Various ones have been developed for various purposes.

Examples:

- Clause form for logical expressions to be used in resolution theorem proving
- Disjunctive normal form for database queries so that they can be entered in a query by example grid.
- Various normal forms for grammars to support specific parsing techniques.

### Clause Form for Logical Expressions

$\forall x : [\text{Roman}(x) \wedge \text{know}(x, \text{Marcus})] \rightarrow [\text{hate}(x, \text{Caesar}) \vee (\forall y : \exists z : \text{hate}(y, z) \rightarrow \text{thinkcrazy}(x, y))]$

becomes

$\neg\text{Roman}(x) \vee \neg\text{know}(x, \text{Marcus}) \vee \text{hate}(x, \text{Caesar}) \vee \neg\text{hate}(y, z) \vee \text{thinkcrazy}(x, z)$

### Disjunctive Normal Form for Queries

(category = fruit or category = vegetable)  
and  
(supplier = A or supplier = B)

becomes

(category = fruit and supplier = A)                    or  
 (category = fruit and supplier = B)                    or  
 (category = vegetable and supplier = A)                or  
 (category = vegetable and supplier = B)

Category	Supplier	Price
fruit	A	
fruit	B	
vegetable	A	
vegetable	B	

### Normal Forms for Grammars

Two of the most common are:

- **Chomsky Normal Form**, in which all rules are of one of the following two forms:
  - $X \rightarrow a$ , where  $a \in \Sigma$ , or
  - $X \rightarrow BC$ , where B and C are nonterminals in G
- **Greibach Normal Form**, in which all rules are of the following form:
  - $X \rightarrow a \beta$ , where  $a \in \Sigma$  and  $\beta$  is a (possibly empty) string of nonterminals

If L is a context-free language that does not contain  $\epsilon$ , then if G is a grammar for L, G can be rewritten into both of these normal forms.

## What Are Normal Forms Good For?

Examples:

- **Chomsky Normal Form:**

- $X \rightarrow a$ , where  $a \in \Sigma$ , or
- $X \rightarrow BC$ , where B and C are nonterminals in G

◆ The branching factor is precisely 2. Tree building algorithms can take advantage of that.

- **Greibach Normal Form**

- $X \rightarrow a\beta$ , where  $a \in \Sigma$  and  $\beta$  is a (possibly empty) string of nonterminals

◆ Precisely one nonterminal is generated for each rule application. This means that we can put a bound on the number of rule applications in any successful derivation.

### Conversion to Chomsky Normal Form

Let G be a grammar for the context-free language L where  $\epsilon \notin L$ .

We construct G', an equivalent grammar in Chomsky Normal Form by:

0. Initially, let  $G' = G$ .

1. Remove from G' all  $\epsilon$  productions:

- 1.1. If there is a rule  $A \rightarrow \alpha B \beta$  and B is nullable, add the rule  $A \rightarrow \alpha \beta$  and delete the rule  $B \rightarrow \epsilon$ .

Example:

$S \rightarrow aA$   
 $A \rightarrow B \mid CD$   
 $B \rightarrow \epsilon$   
 $B \rightarrow a$   
 $C \rightarrow BD$   
 $D \rightarrow b$   
 $D \rightarrow \epsilon$

### Conversion to Chomsky Normal Form

2. Remove from G' all unit productions (rules of the form  $A \rightarrow B$ , where B is a nonterminal):

- 2.1. Remove from G' all unit productions of the form  $A \rightarrow A$ .
- 2.2. For all nonterminals A, find all nonterminals B such that  $A \Rightarrow^* B$ ,  $A \neq B$ .
- 2.3. Create G'' and add to it all rules in G' that are not unit productions.
- 2.4. For all A and B satisfying 3.2, add to G''  
 $A \rightarrow y_1 \mid y_2 \mid \dots$  where  $B \rightarrow y_1 \mid y_2 \mid \dots$  is in G''.
- 2.5. Set G' to G''.

Example:

$A \rightarrow a$   
 $A \rightarrow B$   
 $A \rightarrow EF$   
 $B \rightarrow A$   
 $B \rightarrow CD$   
 $B \rightarrow C$   
 $C \rightarrow ab$

At this point, all rules whose right hand sides have length 1 are in Chomsky Normal Form.



### Conversion to Chomsky Normal Form

3. Remove from  $G'$  all productions  $P$  whose right hand sides have length greater than 1 and include a terminal (e.g.,  $A \rightarrow aB$  or  $A \rightarrow BaC$ ):
  - 3.1. Create a new nonterminal  $T_a$  for each terminal  $a$  in  $\Sigma$ .
  - 3.2. Modify each production  $P$  by substituting  $T_a$  for each terminal  $a$ .
  - 3.3. Add to  $G'$ , for each  $T_a$ , the rule  $T_a \rightarrow a$

Example:

$A \rightarrow aB$   
 $A \rightarrow BaC$   
 $A \rightarrow BbC$

$T_a \rightarrow a$   
 $T_b \rightarrow b$

### Conversion to Chomsky Normal Form

4. Remove from  $G'$  all productions  $P$  whose right hand sides have length greater than 2 (e.g.,  $A \rightarrow BCDE$ )
  - 4.1. For each  $P$  of the form  $A \rightarrow N_1N_2N_3N_4 \dots N_n$ ,  $n > 2$ , create new nonterminals  $M_2, M_3, \dots, M_{n-1}$ .
  - 4.2. Replace  $P$  with the rule  $A \rightarrow N_1M_2$ .
  - 4.3. Add the rules  $M_2 \rightarrow N_2M_3, M_3 \rightarrow N_3M_4, \dots, M_{n-1} \rightarrow N_{n-1}N_n$

Example:

$A \rightarrow BCDE$  ( $n = 4$ )

$A \rightarrow BM_2$   
 $M_2 \rightarrow CM_3$   
 $M_3 \rightarrow DE$

# Top Down Parsing

Read K & S 3.8.

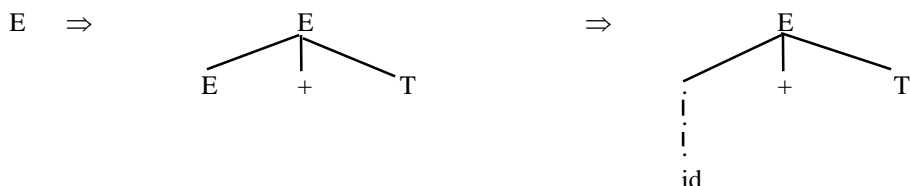
Read Supplementary Materials: Context-Free Languages and Pushdown Automata: Parsing, Sections 1 and 2.

Do Homework 15.

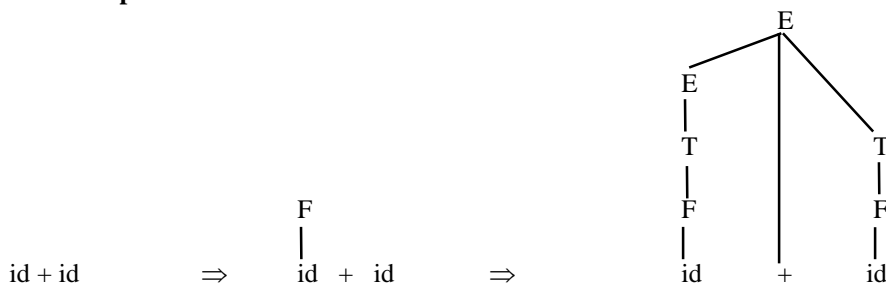
## Parsing

Two basic approaches:

### Top Down



### Bottom Up



## A Simple Parsing Example

A simple top-down parser for arithmetic expressions, given the grammar

- [1]  $E \rightarrow E + T$
- [2]  $E \rightarrow T$
- [3]  $T \rightarrow T * F$
- [4]  $T \rightarrow F$
- [5]  $F \rightarrow (E)$
- [6]  $F \rightarrow id$
- [7]  $F \rightarrow id(E)$

A PDA that does a top down parse:

- (0)  $(1, \epsilon, \epsilon), (2, E)$
- (1)  $(2, \epsilon, E), (2, E+T)$
- (2)  $(2, \epsilon, E), (2, T)$
- (3)  $(2, \epsilon, T), (2, T * F)$
- (4)  $(2, \epsilon, T), (2, F)$
- (5)  $(2, \epsilon, F), (2, (E))$
- (6)  $(2, \epsilon, F), (2, id)$
- (7)  $(2, \epsilon, F), (2, id(E))$
- (8)  $(2, id, id), (2, \epsilon)$
- (9)  $(2, (, ( ), (2, \epsilon)$
- (10)  $(2, ), ) , (2, \epsilon)$
- (11)  $(2, +, +), (2, \epsilon)$
- (12)  $(2, *, *), (2, \epsilon)$

## How Does It Work?

Example:  $\text{id} + \text{id} * \text{id}(\text{id})$

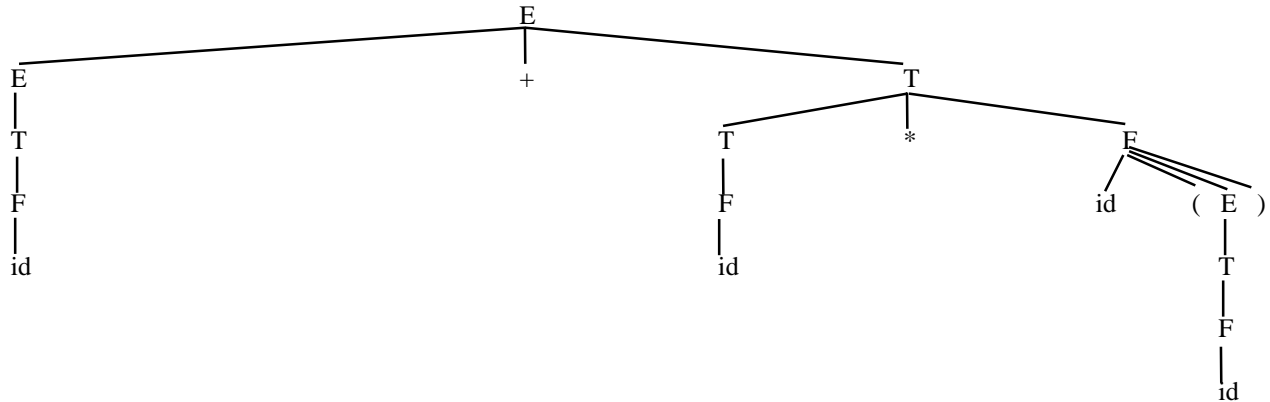
Stack:



## What Does It Produce?

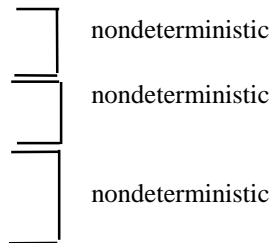
The leftmost derivation of the string. Why?

$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow \text{id} + T \Rightarrow$   
 $\text{id} + T * F \Rightarrow \text{id} + F * F \Rightarrow \text{id} + \text{id} * F \Rightarrow$   
 $\text{id} + \text{id} * \text{id}(E) \Rightarrow \text{id} + \text{id} * \text{id}(T) \Rightarrow$   
 $\text{id} + \text{id} * \text{id}(F) \Rightarrow \text{id} + \text{id} * \text{id}(\text{id})$



## But the Process Isn't Deterministic

- (0) (1,  $\epsilon$ ,  $\epsilon$ ), (2, E)
- (1) (2,  $\epsilon$ , E), (2, E+T)
- (2) (2,  $\epsilon$ , E), (2, T)
- (3) (2,  $\epsilon$ , T), (2, T\*F)
- (4) (2,  $\epsilon$ , T), (2, F)
- (5) (2,  $\epsilon$ , F), (2, (E) )
- (6) (2,  $\epsilon$ , F), (2, id)
- (7) (2,  $\epsilon$ , F), (2, id(E))
- (8) (2, id, id), (2,  $\epsilon$ )
- (9) (2, (, ( ), (2,  $\epsilon$ )
- (10) (2, ), ) ), (2,  $\epsilon$ )
- (11) (2, +, +), (2,  $\epsilon$ )
- (12) (2, \*, \*), (2,  $\epsilon$ )



## Is Nondeterminism A Problem?

Yes.

In the case of regular languages, we could cope with nondeterminism in either of two ways:

- Create an equivalent deterministic recognizer (FSM)
- Simulate the nondeterministic FSM in a number of steps that was still linear in the length of the input string.

For context-free languages, however,

- The best straightforward general algorithm for recognizing a string is  $O(n^3)$  and the best (very complicated) algorithm is based on a reduction to matrix multiplication, which may get close to  $O(n^2)$ .

We'd really like to find a deterministic parsing algorithm that could run in time proportional to the length of the input string.

## Is It Possible to Eliminate Nondeterminism?

In this case: Yes

In general: No

Some definitions:

- A PDA  $M$  is **deterministic** if it has no two transitions such that for some (state, input, stack sequence) the two transitions could both be taken.
- A language  $L$  is **deterministic context-free** if  $L\$ = L(M)$  for some deterministic PDA  $M$ .

**Theorem:** The class of deterministic context-free languages is a *proper* subset of the class of context-free languages.

**Proof:** Later.

## Adding a Terminator to the Language

We define the class of deterministic context-free languages with respect to a terminator ( $\$$ ) because we want that class to be as large as possible.

**Theorem:** Every deterministic CFL (as just defined) is a context-free language.

**Proof:**

Without the terminator ( $\$$ ), many seemingly deterministic cfls aren't. Example:

$$a^* \cup \{a^n b^n : n > 0\}$$

## Possible Solutions to the Nondeterminism Problem

- 1) **Modify the language**
  - Add a terminator  $\$$
- 2) **Change the parsing algorithm**
- 3) **Modify the grammar**

## Modifying the Parsing Algorithm

What if we add the ability to look one character ahead in the input string?

Example:  $id + id * id(id)$

↑↑

$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow id + T \Rightarrow$   
 $id + T * F \Rightarrow id + F * F \Rightarrow id + id * F$

Considering transitions:

- (5)  $(2, \epsilon, F), (2, (E))$
- (6)  $(2, \epsilon, F), (2, id)$
- (7)  $(2, \epsilon, F), (2, id(E))$

If we add to the state an indication of what character is next, we have:

- (5)  $(2, (, \epsilon, F), (2, (E))$
- (6)  $(2, id, \epsilon, F), (2, id)$
- (7)  $(2, id, \epsilon, F), (2, id(E))$

## Modifying the Language

So we've solved part of the problem. But what do we do when we come to the end of the input? What will be the state indicator then?

The solution is to modify the language. Instead of building a machine to accept  $L$ , we will build a machine to accept  $L\$$ .

## Using Lookahead

		(0) $(1, \epsilon, \epsilon), (2, E)$
[1]	$E \rightarrow E + T$	(1) $(2, \epsilon, E), (2, E+T)$
[2]	$E \rightarrow T$	(2) $(2, \epsilon, E), (2, T)$
[3]	$T \rightarrow T * F$	(3) $(2, \epsilon, T), (2, T * F)$
[4]	$T \rightarrow F$	(4) $(2, \epsilon, T), (2, F)$
[5]	$F \rightarrow (E)$	(5) $(2, (, \epsilon, F), (2, (E))$
[6]	$F \rightarrow id$	(6) $(2, id, \epsilon, F), (2, id)$
[7]	$F \rightarrow id(E)$	(7) $(2, id, \epsilon, F), (2, id(E))$
		(8) $(2, id, id), (2, \epsilon)$
		(9) $(2, (, ( ), (2, \epsilon)$
		(10) $(2, ), ) ), (2, \epsilon)$
		(11) $(2, +, +), (2, \epsilon)$
		(12) $(2, *, *), (2, \epsilon)$

For now, we'll ignore the issue of when we read the lookahead character and the fact that we only care about it if the top symbol on the stack is  $F$ .

## Possible Solutions to the Nondeterminism Problem

- 1) **Modify the language**
  - Add a terminator  $\$$
- 2) **Change the parsing algorithm**
  - Add one character look ahead
- 3) **Modify the grammar**

## Modifying the Grammar

Getting rid of identical first symbols:

[6]	$F \rightarrow id$	$(6) (2, id, \epsilon, F), (2, id)$
[7]	$F \rightarrow id(E)$	$(7) (2, id, \epsilon, F), (2, id(E))$

Replace with:

[6']	$F \rightarrow id A$	$(6') (2, id, \epsilon, F), (2, id A)$
[7']	$A \rightarrow \epsilon$	$(7') (2, \epsilon, A), (2, \epsilon)$
[8']	$A \rightarrow (E)$	$(8') (2, (, \epsilon, A), (2, (E))$

The general rule for **left factoring**:

Whenever

$A \rightarrow \alpha\beta_1$
$A \rightarrow \alpha\beta_2 \dots$
$A \rightarrow \alpha\beta_n$

are rules with  $\alpha \neq \epsilon$  and  $n \geq 2$ , then replace them by the rules:

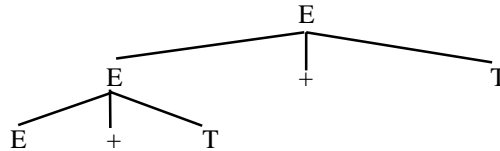
$A \rightarrow \alpha A'$
$A' \rightarrow \beta_1$
$A' \rightarrow \beta_2 \dots$
$A' \rightarrow \beta_n$

## Modifying the Grammar

Getting rid of left recursion:

[1]	$E \rightarrow E + T$	$(1) (2, \epsilon, E), (2, E+T)$
[2]	$E \rightarrow T$	$(2) (2, \epsilon, E), (2, T)$

The problem:



Replace with:

[1]	$E \rightarrow T E'$	$(1) (2, \epsilon, E), (2, T E')$
[2]	$E' \rightarrow + T E'$	$(2) (2, \epsilon, E'), (2, + T E')$
[3]	$E' \rightarrow \epsilon$	$(3) (2, \epsilon, E'), (2, \epsilon)$

## Getting Rid of Left Recursion

The general rule for eliminating **left recursion**:

If  $G$  contains the following rules:

$$A \rightarrow A\alpha_1$$

$$A \rightarrow A\alpha_2 \dots$$

$$A \rightarrow A\alpha_3$$

$$A \rightarrow A\alpha_n$$

$$A \rightarrow \beta_1 \text{ (where } \beta\text{'s do not start with } A\alpha\text{)}$$

$$A \rightarrow \beta_2$$

...

$$A \rightarrow \beta_m$$

Replace them with:

$$A' \rightarrow \alpha_1 A'$$

$$A' \rightarrow \alpha_2 A' \dots$$

$$A' \rightarrow \alpha_3 A'$$

$$A' \rightarrow \alpha_n A'$$

$$A' \rightarrow \epsilon$$

$$A \rightarrow \beta_1 A'$$

$$A \rightarrow \beta_2 A'$$

...

$$A \rightarrow \beta_m A'$$

and  $n > 0$ , then

### Possible Solutions to the Nondeterminism Problem

- I. **Modify the language**
  - A. Add a terminator  $\$$
- II. **Change the parsing algorithm**
  - A. Add one character look ahead
- III. **Modify the grammar**
  - A. Left factor
  - B. Get rid of left recursion

### LL(k) Languages

We have just offered heuristic rules for getting rid of some nondeterminism.

We know that not all context-free languages are deterministic, so there are some languages for which these rules won't work.

We define a **grammar** to be **LL(k)** if it is possible to decide what production to apply by looking ahead at most  $k$  symbols in the input string.

Specifically, a **grammar**  $G$  is **LL(1)** iff, whenever

$A \rightarrow \alpha \mid \beta$  are two rules in  $G$ :

1. For no terminal  $a$  do  $\alpha$  and  $\beta$  derive strings beginning with  $a$ .
2. At most one of  $\alpha \mid \beta$  can derive  $\epsilon$ .
3. If  $\beta \Rightarrow^* \epsilon$ , then  $\alpha$  does not derive any strings beginning with a terminal in  $\text{FOLLOW}(A)$ , defined to be the set of terminals that can immediately follow  $A$  in some sentential form.

We define a **language** to be **LL(k)** if there exists an **LL(k)** grammar for it.

## Implementing an LL(1) Parser

If a language  $L$  has an LL(1) grammar, then we can build a deterministic LL(1) parser for it. Such a parser scans the input Left to right and builds a Leftmost derivation.

The heart of an LL(1) parser is the parsing table, which tells it which production to apply at each step.

For example, here is the parsing table for our revised grammar of arithmetic expressions without function calls:

$V \setminus \Sigma$	id	+	*	(	)	\$
<b>E</b>	$E \rightarrow TE'$			$E \rightarrow TE'$		
<b>E'</b>		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
<b>T</b>	$T \rightarrow FT'$			$T \rightarrow FT'$		
<b>T'</b>		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
<b>F</b>	$F \rightarrow id$			$F \rightarrow (E)$		

Given input  $id + id * id$ , the first few moves of this parser will be:

	E	id + id * id\$
E $\rightarrow$ TE'	TE'	id + id * id\$
T $\rightarrow$ FT'	FT'E'	id + id * id\$
F $\rightarrow$ id	idT'E'	id + id * id\$
	T'E'	+ id * id\$
T' $\rightarrow$ $\epsilon$	E'	+ id * id\$

### But What If We Need a Language That Isn't LL(1)?

Example:

$ST \rightarrow \text{if } C \text{ then } ST \text{ else } ST$   
 $ST \rightarrow \text{if } C \text{ then } ST$

We can apply left factoring to yield:

$ST \rightarrow \text{if } C \text{ then } ST S'$   
 $S' \rightarrow \text{else } ST \mid \epsilon$

Now we've procrastinated the decision. But the language is still ambiguous. What if the input is

$\underline{\text{if } C_1 \text{ then } \underline{\text{if } C_2 \text{ then } ST_1 \text{ else } ST_2}}$

Which bracketing (rule) should we choose?

A common practice is to choose  $S' \rightarrow \text{else } ST$

We can force this if we create the parsing table by hand.

### Possible Solutions to the Nondeterminism Problem

- I. **Modify the language**
  - A. Add a terminator \$
- II. **Change the parsing algorithm**
  - A. Add one character look ahead
  - B. Use a parsing table
  - C. Tailor parsing table entries by hand
- III. **Modify the grammar**
  - A. Left factor
  - B. Get rid of left recursion



## The Price We Pay

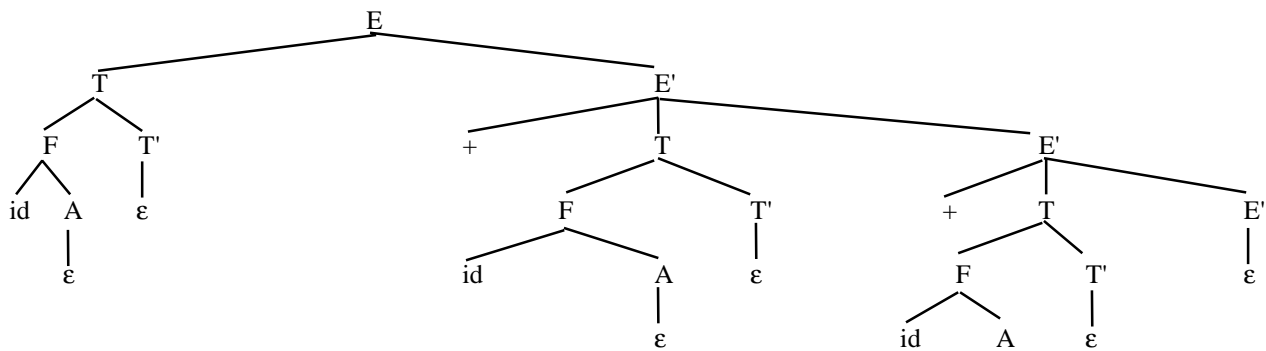
### Old Grammar

- [1]  $E \rightarrow E + T$
- [2]  $E \rightarrow T$
  
- [3]  $T \rightarrow T * F$
- [4]  $T \rightarrow F$
  
- [5]  $F \rightarrow (E)$
- [6]  $F \rightarrow id$
- [7]  $F \rightarrow id(E)$

### New Grammar

- $E \rightarrow TE'$
- $E' \rightarrow +TE'$
- $E' \rightarrow \epsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT'$
- $T' \rightarrow \epsilon$
- $F \rightarrow (E)$
- $F \rightarrow idA$
- $A \rightarrow \epsilon$
- $A \rightarrow (E)$

input = id + id + id



## Comparing Regular and Context-Free Languages

### Regular Languages

- regular exprs.  
or
- regular grammars
- = DFSAs
- recognize
- minimize FSAs

### Context-Free Languages

- context-free grammars
  
- = NDPDAs
- parse
- find deterministic grammars
- find efficient parsers

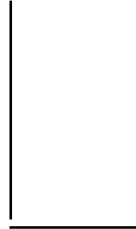
# Bottom Up Parsing

Read Supplementary Materials: Context-Free Languages and Pushdown Automata: Parsing, Section 3.

## Bottom Up Parsing

An Example:

- [1]  $E \rightarrow E + T$
- [2]  $E \rightarrow T$
- [3]  $T \rightarrow T * F$
- [4]  $T \rightarrow F$
- [5]  $F \rightarrow (E)$
- [6]  $F \rightarrow id$



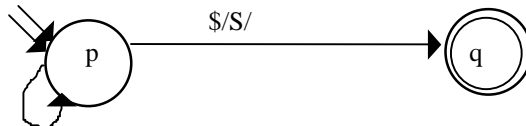
id      +      id      \*      id      \$

## Creating a Bottom Up PDA

There are two basic actions:

1. Shift an input symbol onto the stack
2. Reduce a string of stack symbols to a nonterminal

M will be:



So, to construct M from a grammar G, we need the following transitions:

(1) The shift transitions:

$$((p, a, \epsilon), (p, a)), \text{ for each } a \in \Sigma$$

(2) The reduce transitions:

$$((p, \epsilon, \alpha^R), (p, A)), \text{ for each rule } A \rightarrow \alpha \text{ in } G.$$

(3) The finish up transition (accept):

$$((p, \$, S), (q, \epsilon))$$

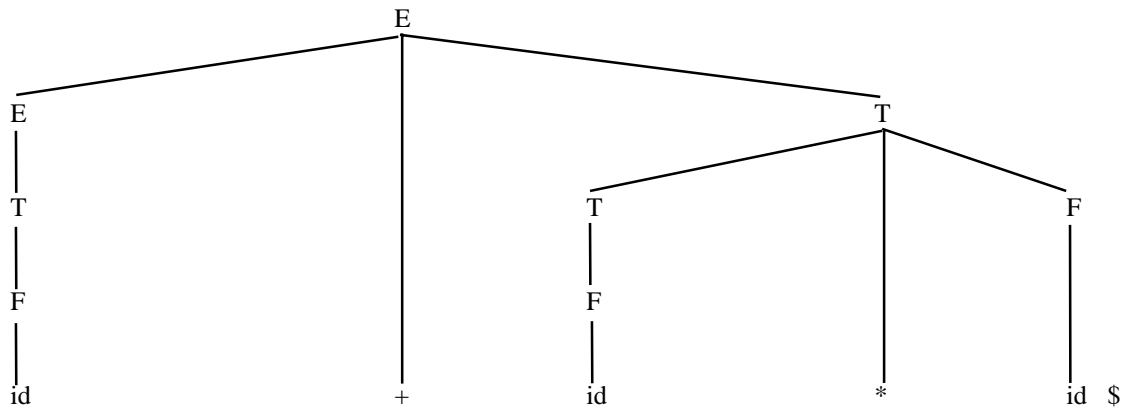
(This is the “bottom-up” CFG to PDA conversion algorithm.)

## M for Expressions

0	(p, a, ε), (p, a) for each a ∈ Σ
1	(p, ε, T + E), (p, E)
2	(p, ε, T), (p, E)
3	(p, ε, F * T), (p, T)
4	(p, ε, F), (p, T)
5	(p, ε, "("E"("), (p, F)
6	(p, ε, id), (p, F)
7	(p, \$, E), (q, ε)

<i>trans (action)</i>	<i>state</i>	<i>unread input</i>	<i>stack</i>
	p	id + id * id\$	ε
0 (shift)	p	+ id * id\$	id
6 (reduce F → id)	p	+ id * id\$	F
4 (reduce T → F)	p	+ id * id\$	T
2 (reduce E → T)	p	+ id * id\$	E
0 (shift)	p	id * id\$	+E
0 (shift)	p	* id\$	id+E
6 (reduce F → id)	p	* id\$	F+E
4 (reduce T → F)	p	* id\$	T+E (could also reduce)
0 (shift)	p	id\$	*T+E
0 (shift)	p	\$	id*T+E
6 (reduce F → id)	p	\$	F*T+E (could also reduce T → F)
3 (reduce T → T * F)	p	\$	T+E
1 (reduce E → E + T)	p	\$	E
7 (accept)	q	\$	ε

### The Parse Tree



### Producing the Rightmost Derivation

We can reconstruct the derivation that we found by reading the results of the parse bottom to top, producing:

E ⇒	E+ id* id⇒
E+ T ⇒	T+ id*id⇒
E+ T* F⇒	F+ id*id⇒
E+ T* id⇒	id+ id*id
E+ F* id⇒	

This is exactly the rightmost derivation of the input string.

## Possible Solutions to the Nondeterminism Problem

- 1) **Modify the language**
  - Add a terminator \$
- 2) **Change the parsing algorithm**
  - *Add one character look ahead*
  - *Use a parsing table*
  - *Tailor parsing table entries by hand*
  - **Switch to a bottom-up parser**
- 3) **Modify the grammar**
  - *Left factor*
  - *Get rid of left recursion*

### Solving the Shift vs. Reduce Ambiguity With a Precedence Relation

Let's return to the problem of deciding when to shift and when to reduce (as in our example).

We chose, correctly, to shift \* onto the stack, instead of reducing T+E to E.

This corresponds to knowing that "+" has low precedence, so if there are any other operations, we need to do them first.

Solution:

1. Add a one character lookahead capability.
2. Define the precedence relation

$$P \subseteq \begin{array}{ccc} ( & V & \times \\ \text{top} & & \text{next} \\ \text{stack} & & \text{input} \\ \text{symbol} & & \text{symbol} \end{array} \{ \Sigma \cup \$ \} )$$

If (a,b) is in P, we reduce (without consuming the input) . Otherwise we shift (consuming the input).

### How Does It Work?

We're reconstructing rightmost derivations backwards. So suppose a rightmost derivation contains

$$\begin{array}{c} \beta\gamma abx \\ \uparrow \\ \beta Abx \\ \uparrow^* \\ S \end{array} \quad \leftarrow \text{corresponding to a rule } A \rightarrow \gamma a \text{ and not some rule } X \rightarrow ab$$

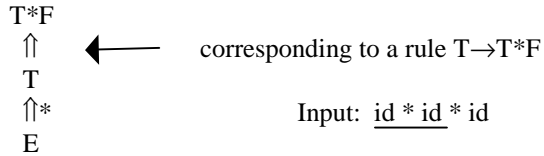
We want to undo rule A. So if the top of the stack is

$$\begin{array}{|c|} \hline a \\ \hline \gamma \\ \hline \end{array}$$

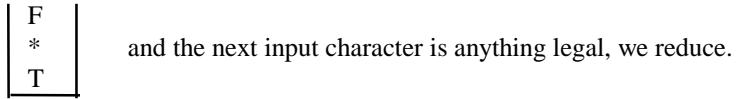
and the next input character is b, we reduce now, before we put the b on the stack.

To make this happen, we put (a, b) in P. That means we'll try to reduce if a is on top of the stack and b is the next character. We will actually succeed if the next part of the stack is  $\gamma$ .

### Example



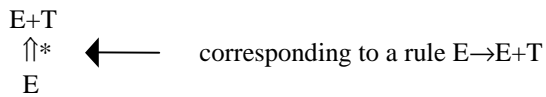
We want to undo rule T. So if the top of the stack is



The precedence relation for expressions:

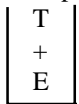
V \ Σ	(	)	id	+	*	\$
(						
)		•		•	•	•
id		•		•	•	•
+						
*						
E						
T		•		•		•
F		•		•	•	•

### A Different Example



We want to undo rule E if the input is E + T \$  
 or E + T + id  
 but not E + T \* id

The top of the stack is



The precedence relation for expressions:

V \ Σ	(	)	id	+	*	\$
(						
)		•		•	•	•
id		•		•	•	•
+						
*						
E						
T		•		•		•
F		•		•	•	•

## What About If Then Else?

ST  $\rightarrow$  if C then ST else ST  
 ST  $\rightarrow$  if C then ST

What if the input is

$\overline{\text{if } C_1 \text{ then } \overline{\text{if } C_2 \text{ then } ST_1 \text{ else } ST_2}}$   
 $\uparrow_1 \qquad \qquad \uparrow_2$

Which bracketing (rule) should we choose?

We don't put (ST, else) in the precedence relation, so we will not reduce at 1. At 2, we reduce:

ST2	2
else	
ST1	1
then	
C2	
if	
then	
C1	
if	

### Resolving Reduce vs. Reduce Ambiguities

- 0 (p, a,  $\epsilon$ ), (p, a) for each  $a \in \Sigma$
- 1 (p,  $\epsilon$ , T + E), (p, E)
- 2 (p,  $\epsilon$ , T), (p, E)
- 3 (p,  $\epsilon$ , F \* T), (p, T)
- 4 (p,  $\epsilon$ , F), (p, T)
- 5 (p,  $\epsilon$ , "(") E "("), (p, F)
- 6 (p,  $\epsilon$ , id), (p, F)
- 7 (p, \$, E), (q,  $\epsilon$ )

<i>trans (action)</i>	<i>state</i>	<i>unread input</i>	<i>stack</i>
	p	id + id * id\$	$\epsilon$
0 (shift)	p	+ id * id\$	id
6 (reduce F $\rightarrow$ id)	p	+ id * id\$	F
4 (reduce T $\rightarrow$ F)	p	+ id * id\$	T
2 (reduce E $\rightarrow$ T)	p	+ id * id\$	E
0 (shift)	p	id * id\$	+E
0 (shift)	p	* id\$	id+E
6 (reduce F $\rightarrow$ id)	p	* id\$	F+E
4 (reduce T $\rightarrow$ F)	p	* id\$	T+E (could also reduce)
0 (shift)	p	id\$	*T+E
0 (shift)	p	\$	id*T+E
6 (reduce F $\rightarrow$ id)	p	\$	F*T+E (could also reduce T $\rightarrow$ F)
3 (reduce T $\rightarrow$ T * F)	p	\$	T+E
1 (reduce E $\rightarrow$ E + T)	p	\$	E
7 (accept)	q	\$	$\epsilon$

## The Longest Prefix Heuristic

A simple to implement heuristic rule, when faced with competing reductions, is:

*Choose the longest possible stack string to reduce.*

Example:

Suppose the stack has  $\frac{\text{T}}{\text{F} * \text{T}} + \text{E}$   
                                  ↑  
                                  T  
                                  ↓  
                                  T

We call grammars that become unambiguous with the addition of a precedence relation and the longest string reduction heuristic **weak precedence grammars**.

### Possible Solutions to the Nondeterminism Problem in a Bottom Up Parser

- 1) **Modify the language**
  - Add a terminator \$
- 2) **Change the parsing algorithm**
  - Add one character lookahead
  - Use a precedence table
  - Add the longest first heuristic for reduction
  - **Use an LR parser**
- 3) **Modify the grammar**

## LR Parsers

LR parsers scan each input **L**eft to right and build a **R**ightmost derivation. They operate bottom up and deterministically using a parsing table derived from a grammar for the language to be recognized.

A grammar that can be parsed by an LR parser examining up to k input symbols on each move is an **LR(k)** grammar. Practical LR parsers set k to 1.

An LALR ( or Look Ahead LR) parser is a specific kind of LR parser that has two desirable properties:

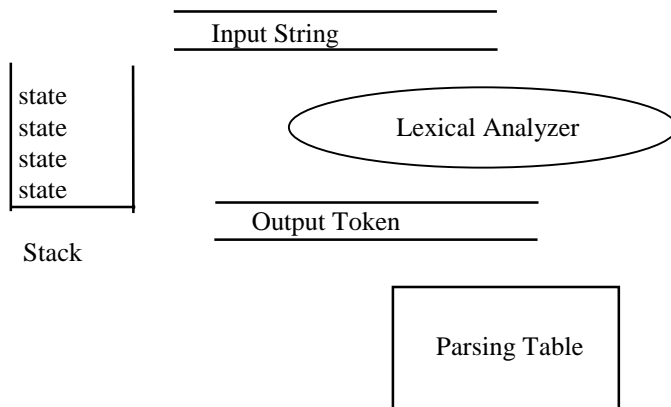
- The parsing table is not huge.
- Most useful languages can be parsed.

Another big reason to use an LALR parser:

There are automatic tools that will construct the required parsing table from a grammar and some optional additional information.

We will be using such a tool: **yacc**

## How an LR Parser Works



In simple cases, think of the "states" on the stack as corresponding to either terminal or nonterminal characters.

In more complicated cases, the states contain more information: they encode both the top stack symbol and some facts about lower objects in the stack. This information is used to determine which action to take in situations that would otherwise be ambiguous.

### The Actions the Parser Can Take

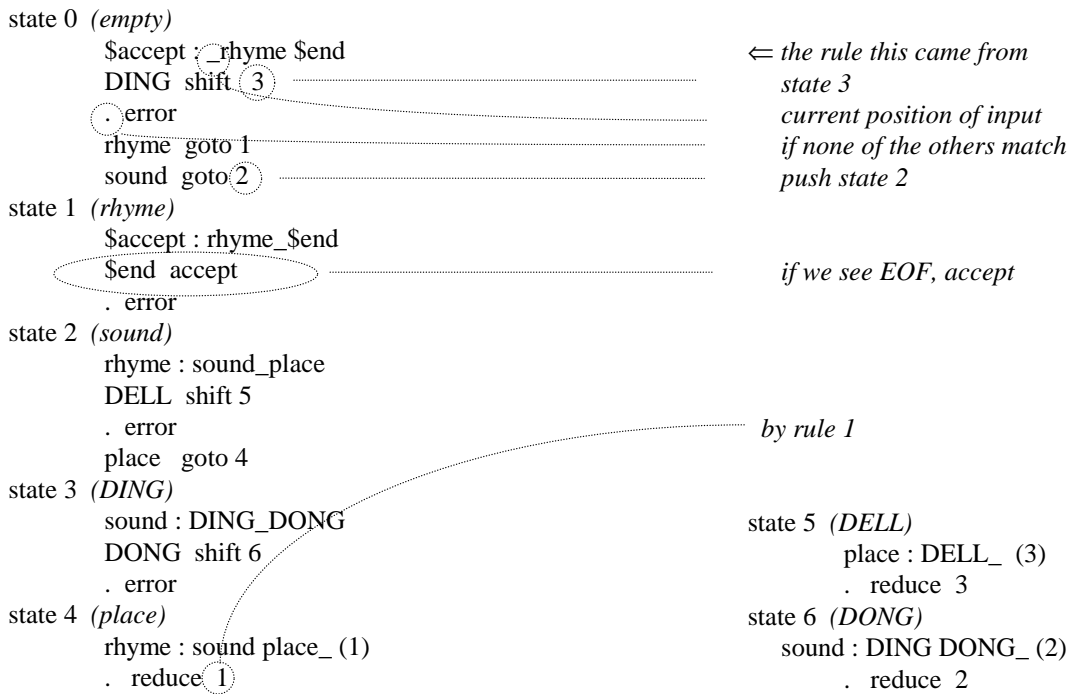
At each step of its operation, an LR parser does the following two things:

- 1) Based on its current state, it decides whether it needs a lookahead token. If it does, it gets one.
- 2) Based on its current state and the lookahead token if there is one, it chooses one of four possible actions:
  - Shift the lookahead token onto the stack and clear the lookahead token.
  - Reduce the top elements of the stack according to some rule of the grammar.
  - Detect the end of the input and accept the input string.
  - Detect an error in the input.



## A Simple Example

- 0: S → rhyme \$end ;
- 1: rhyme → sound place ;
- 2: sound → DING DONG ;
- 3: place → DELL

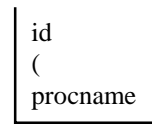


## When the States Are More than Just Stack Symbols

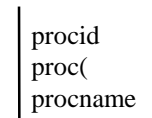
- [1] <stmt> → procname ( <paramlist> )
- [2] <stmt> → <exp> := <exp>
- [3] <paramlist> → <paramlist>, <param> | <param>
- [4 ] <param> → id
- [5] <exp> → arrayname ( <subscriptlist> )
- [6] <subscriptlist> → <subscriptlist>, <sub> | <sub>
- [7] <sub> → id

Example:

procname ( id )  
 ↑



Should we reduce id by rule 4 or rule 7?



The parsing table can get complicated as we incorporate more stack history into the states.

**The Language Interpretation Problem:**

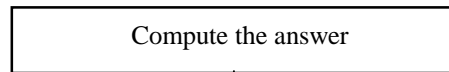
Input:  $-(17 * 83.56) + 72 / 12$



Output: -1414.52

**The Language Interpretation Problem:**

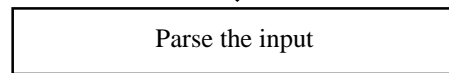
Input:  $-(17 * 83.56) + 72 / 12$



Output: -1414.52

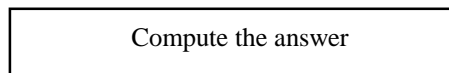
**The Language Interpretation Problem:**

Input:  $-(17 * 83.56) + 72 / 12$



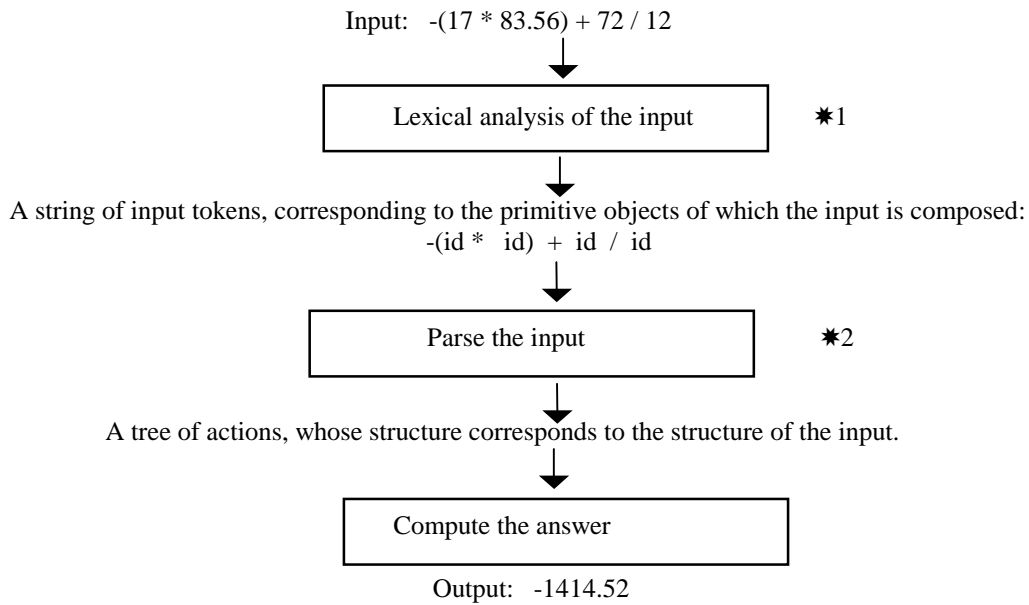
\*2

A tree of actions, whose structure corresponds to the structure of the input.

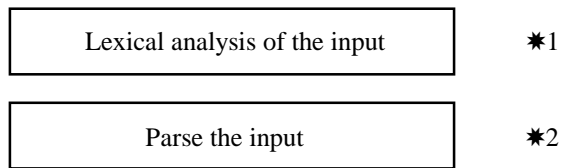


Output: -1414.52

### The Language Interpretation Problem:

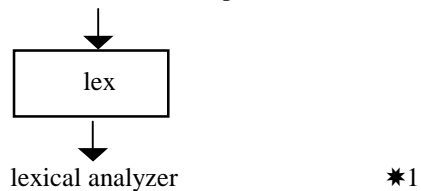


### yacc and lex

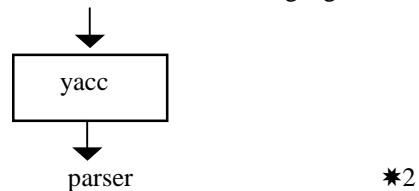


Where do the procedures to do these things come from?

regular expressions that describe patterns



grammar rules and other facts about the language



## lex

The input to lex:        definitions  
                          %%  
                          rules  
                          %%  
                          user routines

All strings that are not matched by any rule are simply copied to the output.

Rules:

```
[ \\t]+;                                get rid of blanks and tabs  
[A-Za-z][A-Za-z0-9]*        return(ID);                        find identifiers  
[0-9]+                {        sscanf(yytext, "%d", &yyval);  
                          return (INTEGER); }                return INTEGER and put the value in yyval
```

### How Does lex Deal with Ambiguity in Rules?

lex invokes two disambiguating rules:

1. The longest match is preferred.
2. Among rules that matched the same number of characters, the rule given first is preferred.

Example:

```
integer    action 1  
[a-z]+    action 2
```

```
input:                integers                take action 2  
                      integer                take action 1
```

## yacc (Yet Another Compiler Compiler)

The input to yacc:

```
declarations  
%%  
rules  
%%  
#include "lex.yy.c"  
any other programs
```

This structure means that lex.yy.c will be compiled as part of y.tab.c, so it will have access to the same token names.

Declarations:

```
%token name1 name2 ...
```

Rules:

```
V        : a b c  
V        : a b c                        {action}  
V        : a b c                        {$$ = $2}                returns the value of b
```

## Example

Input to yacc:

```

%token DING DONG DELL
%%
rhyme :  sound place ;
sound  :  DING DONG ;
place  :  DELL
%%
#include "lex.yy.c"

```

---

<pre> state 0 (<i>empty</i>)   \$accept : _rhyme \$end   DING shift 3   . error   rhyme goto 1   sound goto 2 state 1 (<i>rhyme</i>)   \$accept : rhyme_\$end   \$end accept   . error state 2 (<i>sound</i>)   rhyme : sound_place   DELL shift 5   . error   place goto 4 </pre>	<pre> state 3 (<i>DING</i>)   sound : DING_DONG   DONG shift 6   . error state 4 (<i>place</i>)   rhyme : sound_place_ (1)   . reduce 1 state 5 (<i>DELL</i>)   place : DELL_ (3)   . reduce 3 state 6 (<i>DONG</i>)   sound : DING_DONG_ (2)   . reduce 2 </pre>
--	---

### How Does yacc Deal with Ambiguity in Grammars?

The parser table that yacc creates represents some decision about what to do if there is ambiguity in the input grammar rules. How does yacc make those decisions? By default, yacc invokes two disambiguating rules:

1. In the case of a shift/reduce conflict, shift.
  2. In the case of a reduce/reduce conflict, reduce by the earlier grammar rule.
- yacc tells you when it has had to invoke these rules.

### Shift/Reduce Conflicts - If Then Else

ST → if C then ST else ST  
 ST → if C then ST

What if the input is

```

if C1 then if C2 then ST1 else ST2

```

↑<sub>1</sub>            ↑<sub>2</sub>

Which bracketing (rule) should we choose?

yacc will choose to shift rather than reduce.

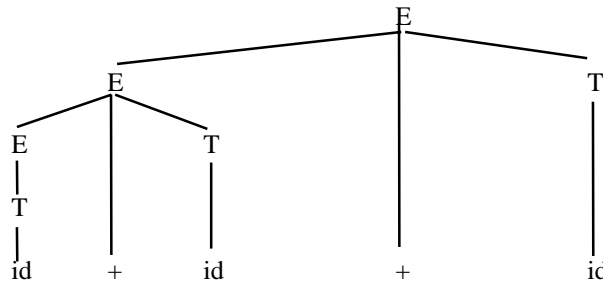
ST2	2
else	
ST1	1
then	
C2	
if	
then	
C1	
if	

### Shift/Reduce Conflicts - Left Associativity

We know that we can force left associativity by writing it into our grammars.

Example:

$E \rightarrow E + T$   
 $E \rightarrow T$   
 $T \rightarrow id$



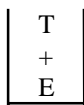
What does the shift rather than reduce heuristic if we instead write:

$E \rightarrow E + E$   
 $E \rightarrow id$

id + id + id

### Shift/Reduce Conflicts - Operator Precedence

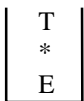
Recall the problem: input: id + id \* id



Should we reduce or shift on \* ?

The "always shift" rule solves this problem.

But what about: id \* id + id



Should we reduce or shift on + ?

This time, if we shift, we'll fail.

One solution was the precedence table, derived from an unambiguous grammar, which can be encoded into the parsing table of an LR parser, since it tells us what to do for each top-of-stack, input character combination.

### Operator Precedence

We know that we can write an unambiguous grammar for arithmetic expressions that gets the precedence right. But it turns out that we can build a faster parser if we instead write:

$E \rightarrow E + E \mid E * E \mid (E) \mid id$

And, in addition, we specify operator precedence. In yacc, we specify associativity (since we might not always want left) and precedence using statements in the declaration section of our grammar:

```
%left '+' '-'
%left '*' '/'
```

Operators on the first line have lower precedence than operators on the second line, and so forth.

## Reduce/Reduce Conflicts

Recall:

2. In the case of a reduce/reduce conflict, reduce by the earlier grammar rule.

This can easily be used to simulate the longest prefix heuristic, "Choose the longest possible stack string to reduce."

```
[1]      E → E + T
[2]      E → T
[3]      T → T * F
[4]      T → F
[5]      F → (E)
[6]      F → id
```

## Generating an Executable System

Step 1: Create the input to lex and the input to yacc.

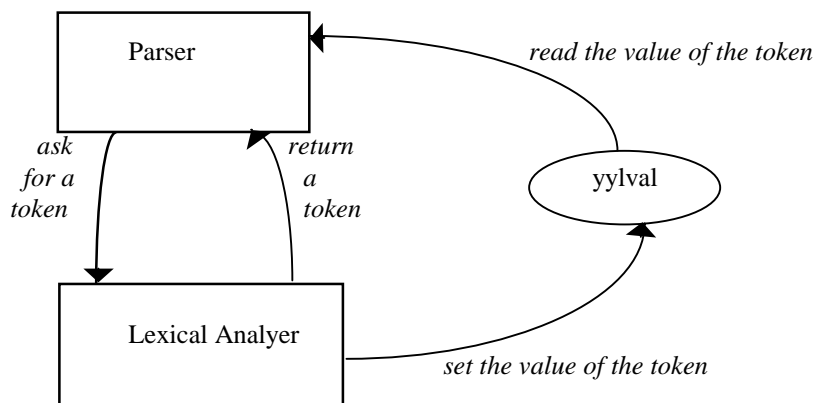
Step 2:

```
$ lex ourlex.l          creates lex.yy.c
$ yacc ouryacc.y        creates y.tab.c
$ cc -o ourprog y.tab.c -ly -ll  actually compiles y.tab.c and lex.yy.c, which is included.
                                -ly links the yacc library, which includes main and yyerror.
                                -ll links the lex library
```

Step 3: Run the program

```
$ ourprog
```

## Runtime Communication Between lex and yacc-Generated Modules



## Summary

Efficient parsers for languages with the complexity of a typical programming language or command line interface:

- Make use of special purpose constructs, like precedence, that are very important in the target languages.
- May need complex transition functions to capture all the relevant history in the stack.
- Use heuristic rules, like shift instead of reduce, that have been shown to work most of the time.
- Would be very difficult to construct by hand (as a result of all of the above).
- Can easily be built using a tool like yacc.

## Languages That Are and Are Not Context-Free

Read K & S 3.5, 3.6, 3.7.

Read Supplementary Materials: Context-Free Languages and Pushdown Automata: Closure Properties of Context-Free Languages

Read Supplementary Materials: Context-Free Languages and Pushdown Automata: The Context-Free Pumping Lemma.  
Do Homework 16.

### Deciding Whether a Language is Context-Free

**Theorem:** There exist languages that are not context-free.

**Proof:**

(1) There are a countably infinite number of context-free languages. This true because every description of a context-free language is of finite length, so there are a countably infinite number of such descriptions.

(2) There are an uncountable number of languages.

Thus there are more languages than there are context-free languages.

So there must exist some languages that are not context-free.

Example:  $\{a^n b^n c^n\}$

### Showing that a Language is Context-Free

Techniques for showing that a language  $L$  is context-free:

1. Exhibit a context-free grammar for  $L$ .
2. Exhibit a PDA for  $L$ .
3. Use the closure properties of context-free languages.

Unfortunately, these are weaker than they are for regular languages.

### The Context-Free Languages are Closed Under Union

Let  $G_1 = (V_1, \Sigma_1, R_1, S_1)$  and

$G_2 = (V_2, \Sigma_2, R_2, S_2)$

Assume that  $G_1$  and  $G_2$  have disjoint sets of nonterminals, not including  $S$ .

Let  $L = L(G_1) \cup L(G_2)$

We can show that  $L$  is context-free by exhibiting a CFG for it:

### The Context-Free Languages are Closed Under Concatenation

Let  $G_1 = (V_1, \Sigma_1, R_1, S_1)$  and

$G_2 = (V_2, \Sigma_2, R_2, S_2)$

Assume that  $G_1$  and  $G_2$  have disjoint sets of nonterminals, not including  $S$ .

Let  $L = L(G_1) L(G_2)$

We can show that  $L$  is context-free by exhibiting a CFG for it:



## The Context-Free Languages are Closed Under Kleene Star

Let  $G_1 = (V_1, \Sigma_1, R_1, S_1)$

Assume that  $G_1$  does not have the nonterminal  $S$ .

Let  $L = L(G_1)^*$

We can show that  $L$  is context-free by exhibiting a CFG for it:

## What About Intersection and Complement?

We know that they share a fate, since

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

But what fate?

We proved closure for regular languages two different ways. Can we use either of them here:

1. Given a deterministic automaton for  $L$ , construct an automaton for its complement. Argue that, if closed under complement and union, must be closed under intersection.
2. Given automata for  $L_1$  and  $L_2$ , construct a new automaton for  $L_1 \cap L_2$  by simulating the parallel operation of the two original machines, using states that are the Cartesian product of the sets of states of the two original machines.

More on this later.

## The Intersection of a Context-Free Language and a Regular Language is Context-Free

$L = L(M_1)$ , a PDA  $= (K_1, \Sigma, \Gamma_1, \Delta_1, s_1, F_1)$

$R = L(M_2)$ , a deterministic FSA  $= (K_2, \Sigma, \delta, s_2, F_2)$

We construct a new PDA,  $M_3$ , that accepts  $L \cap R$  by simulating the parallel execution of  $M_1$  and  $M_2$ .

$M = (K_1 \times K_2, \Sigma, \Gamma_1, \Delta, (s_1, s_2), F_1 \times F_2)$

Insert into  $\Delta$ :

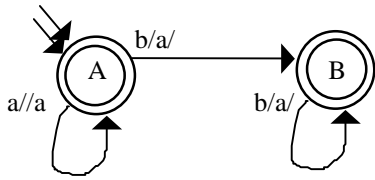
For each rule  $((q_1, a, \beta), (p_1, \gamma))$  in  $\Delta_1$ ,  
and each rule  $(q_2, a, p_2)$  in  $\delta$ ,  
 $((q_1, q_2), a, \beta), ((p_1, p_2), \gamma)$

For each rule  $((q_1, \epsilon, \beta), (p_1, \gamma))$  in  $\Delta_1$ ,  
and each state  $q_2$  in  $K_2$ ,  
 $((q_1, q_2), \epsilon, \beta), ((p_1, q_2), \gamma)$

This works because: we can get away with only one stack.

**Example**

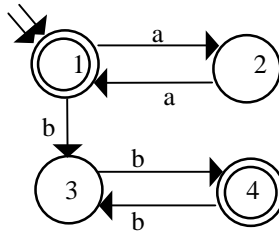
$L = a^n b^n$



- $((A, a, \epsilon), (A, a))$
- $((A, b, a), (B, \epsilon))$
- $((B, b, a), (B, \epsilon))$

A PDA for L:

$(aa)^*(bb)^*$



- $(1, a, 2)$
- $(1, b, 3)$
- $(2, a, 1)$
- $(3, b, 4)$
- $(4, b, 3)$

**Don't Try to Use Closure Backwards**

One Closure Theorem:

If  $L_1$  and  $L_2$  are context free, then so is

$$L_3 = \underline{L_1} \cup \underline{L_2}$$

But what if  $L_3$  and  $L_1$  are context free? What can we say about  $L_2$ ?

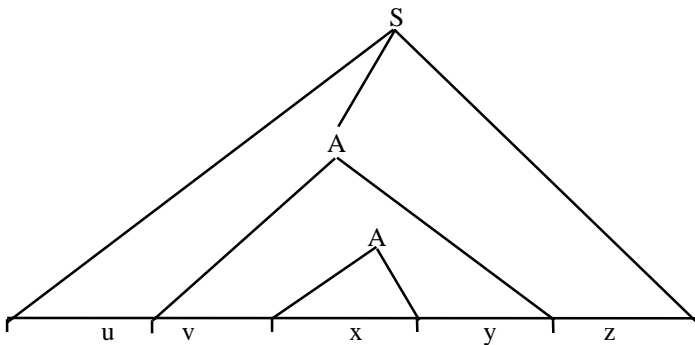
$$\underline{L_3} = \underline{L_1} \cup \underline{L_2}$$

Example:

$$a^n b^n c^* = a^n b^n c^* \cup a^n b^n c^n$$

**The Context-Free Pumping Lemma**

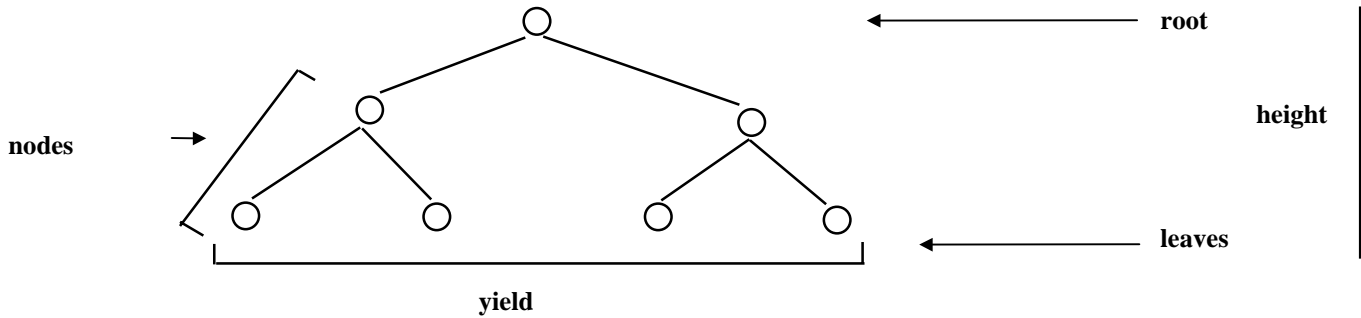
This time we use parse trees, not automata as the basis for our argument.



If  $L$  is a context-free language, and if  $w$  is a string in  $L$  where  $|w| > K$ , for some value of  $K$ , then  $w$  can be rewritten as  $uvxyz$ , where  $|vy| > 0$  and  $|vxy| \leq M$ , for some value of  $M$ .

$uxz, uvxyz, uvvxyyz, uvvvxyyyz$ , etc. (i.e.,  $uv^nxy^n z$ , for  $n \geq 0$ ) are all in  $L$ .

### Some Tree Basics

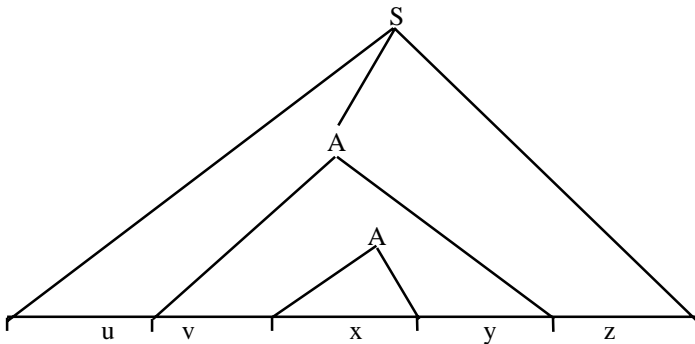


**Theorem:** The length of the yield of any tree  $T$  with height  $H$  and branching factor (**fanout**)  $B$  is  $\leq B^H$ .

**Proof:** By induction on  $H$ . If  $H$  is 1, then just a single rule applies. By definition of fanout, the longest yield is  $B$ . Assume true for  $H = n$ .

Consider a tree with  $H = n + 1$ . It consists of a root, and some number of subtrees, each of which is of height  $\leq n$  (so induction hypothesis holds) and yield  $\leq B^n$ . The number of subtrees  $\leq B$ . So the yield must be  $\leq B(B^n)$  or  $B^{n+1}$ .

### What Is K?



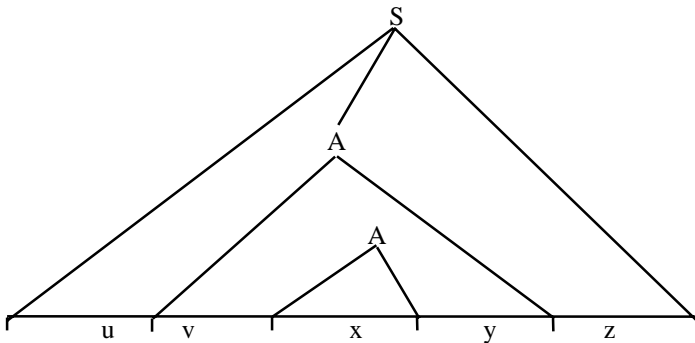
Let  $T$  be the number of nonterminals in  $G$ .

If there is a tree of height  $> T$ , then some nonterminal occurs more than once on some path. If it does, we can pump its yield.

Since a tree of height  $= T$  can produce only strings of length  $\leq B^T$ , any string of length  $> B^T$  must have a repeated nonterminal and thus be pumpable.

So  $K = B^T$ , where  $T$  is the number of nonterminals in  $G$  and  $B$  is the branching factor (fanout).

### What is M?



Assume that we are considering the bottom most two occurrences of some nonterminal. Then the yield of the upper one is at most  $B^{T+1}$  (since only one nonterminal repeats).

So  $M = B^{T+1}$ .

## The Context-Free Pumping Lemma

**Theorem:** Let  $G = (V, \Sigma, R, S)$  be a context-free grammar with  $T$  nonterminal symbols and fanout  $B$ . Then any string  $w \in L(G)$  where  $|w| > K (B^T)$  can be rewritten as  $w = uvxyz$  in such a way that:

- $|vy| > 0$ ,
- $|vxy| \leq M (B^{T+1})$ , (making this the "strong" form),
- for every  $n \geq 0$ ,  $uv^nxy^n z$  is in  $L(G)$ .

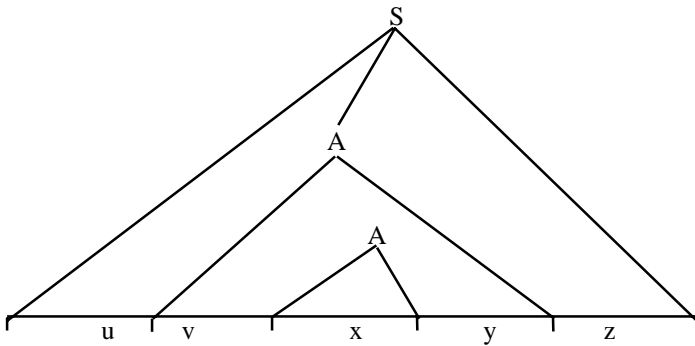
**Proof:**

Let  $w$  be such a string and let  $T$  be the parse tree with root labeled  $S$  and with yield  $w$  that has the smallest number of leaves among all parse trees with the same root and yield.  $T$  has a path of length at least  $T+1$ , with a bottommost repeated nonterminal, which we'll call  $A$ . Clearly  $v$  and  $y$  can be repeated any number of times (including 0). If  $|vy| = 0$ , then there would be a tree with root  $S$  and yield  $w$  with fewer leaves than  $T$ . Finally,  $|vxy| \leq B^{T+1}$ .

### An Example of Pumping

$$L = \{a^n b^n c^n : n \geq 0\}$$

Choose  $w = a^i b^i c^i$  where  $i > \lceil K/3 \rceil$  (making  $|w| > K$ )



Unfortunately, we don't know where  $v$  and  $y$  fall. But there are two possibilities:

1. If  $vy$  contains all three symbols, then at least one of  $v$  or  $y$  must contain two of them. But then  $uvvxyyz$  contains at least one out of order symbol.
2. If  $vy$  contains only one or two of the symbols, then  $uvvxyyz$  must contain unequal numbers of the symbols.

### Using the Strong Pumping Lemma for Context Free Languages

If  $L$  is context free, then

There exist  $K$  and  $M$  (with  $M \geq K$ ) such that

For all strings  $w$ , where  $|w| > K$ ,

(Since true for all such  $w$ , it must be true for any particular one, so you pick  $w$ )

(Hint: describe  $w$  in terms of  $K$  or  $M$ )

there exist  $u, v, x, y, z$  such that  $w = uvxyz$  and  $|vy| > 0$ , and  
 $|vxy| \leq M$ , and  
 for all  $n \geq 0$ ,  $uv^nxy^n z$  is in  $L$ .

We need to **pick  $w$** , then show that there are no values for  $uvxyz$  that satisfy all the above criteria. To do that, we just need to focus on possible values for  $v$  and  $y$ , the pumpable parts. So we **show that all possible picks for  $v$  and  $y$  violate at least one of the criteria.**

**Write out a single string,  $w$**  (in terms of  $K$  or  $M$ ) **Divide  $w$  into regions.**

For each possibility for  $v$  and  $y$  (described in terms of the regions defined above), find some value  $n$  such that  $uv^nxy^n z$  is not in  $L$ . Almost always, the easiest values are 0 (pumping out) or 2 (pumping in). Your value for  $n$  may differ for different cases.

v

y

n

why the resulting string is not in L

[1]

[2]

[3]

[4]

[5]

[6]

[7]

[8]

[9]

[10]

Convince the reader that there are no other cases.

Q. E. D.

### A Pumping Lemma Proof in Full Detail

Proof that  $L = \{a^n b^n c^n : n \geq 0\}$  is not context free.

Suppose L is context free. The context free pumping lemma applies to L. Let M be the number from the pumping lemma. Choose  $w = a^M b^M c^M$ . Now  $w \in L$  and  $|w| > M \geq K$ . From the pumping lemma, for all strings w, where  $|w| > K$ , there exist u, v, x, y, z such that  $w = uvxyz$  and  $|vy| > 0$ , and  $|vxy| \leq M$ , and for all  $n \geq 0$ ,  $uv^n xy^n z$  is in L. There are two main cases:

1. Either v or y contains two or more different types of symbols (“a”, “b” or “c”). In this case,  $uv^2xy^2z$  is not of the form  $a^*b^*c^*$  and hence  $uv^2xy^2z \notin L$ .
2. Neither v nor y contains two or more different types of symbols. In this case, vy may contain at most two types of symbols. The string  $uv^0xy^0z$  will decrease the count of one or two types of symbols, but not the third, so  $uv^0xy^0z \notin L$ .

Cases 1 and 2 cover all the possibilities. Therefore, regardless of how w is partitioned, there is some  $uv^n xy^n z$  that is not in L. Contradiction. Therefore L is not context free.

Note: the underlined parts of the above proof is “boilerplate” that can be reused. A complete proof should have this text or something equivalent.

### Context-Free Languages Over a Single-Letter Alphabet

**Theorem:** Any context-free language over a single-letter alphabet is regular.

Examples:

$$\begin{aligned}
 L &= \{a^n b^n\} \\
 L' &= \{a^n a^n\} \\
 &= \{a^{2n}\} \\
 &= \{w \in \{a\}^* : |w| \text{ is even}\}
 \end{aligned}$$

$$\begin{aligned}
 L &= \{ww^R : w \in \{a, b\}^*\} \\
 L' &= \{ww^R : w \in \{a\}^*\} \\
 &= \{ww : w \in \{a\}^*\} \\
 &= \{w \in \{a\}^* : |w| \text{ is even}\}
 \end{aligned}$$

$$\begin{aligned}
 L &= \{a^n b^m : n, m \geq 0 \text{ and } n \neq m\} \\
 L' &= \{a^n a^m : n, m \geq 0 \text{ and } n \neq m\} \\
 &=
 \end{aligned}$$

**Proof:** See Parikh's Theorem

### Another Language That Is Not Context Free

$$L = \{a^n : n \geq 1 \text{ is prime}\}$$

Two ways to prove that L is not context free:

1. Use the pumping lemma:

Choose a string  $w = a^n$  such that n is prime and  $n > K$ .

$$w = \underbrace{aaaaaaaaaaaaaaaaaaaaaaaa}_{u} \underbrace{a}_{v} \underbrace{x} \underbrace{aaaaaaaaaaaaaaaa}_{y} \underbrace{a}_{z}$$

Let  $vy = a^p$  and  $uxz = a^f$ . Then  $r + kp$  must be prime for all values of k. This can't be true, as we argued to show that L was not regular.

2.  $|\Sigma_L| = 1$ . So if L were context free, it would also be regular. But we know that it is not. So it is not context free either.

### Using Pumping and Closure

$$L = \{w \in \{a, b, c\}^* : w \text{ has an equal number of a's, b's, and c's}\}$$

L is not context free.

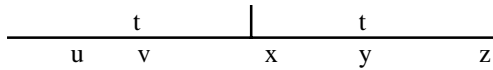
Try pumping: Let  $w = a^K b^K c^K$

Now what?

### Using Intersection with a Regular Language to Make Pumping Tractable

$$L = \{tt : t \in \{a, b\}^*\}$$

Let's try pumping:  $|w| > K$



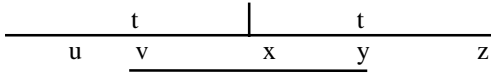
What if u is  $\epsilon$ ,  
 v is w,  
 x is  $\epsilon$ ,  
 y is w, and  
 z is  $\epsilon$

Then all pumping tells us is that  $t^n t^n$  is in L.

$$L = \{tt : t \in \{a, b\}^*\}$$

What if we let  $|w| > M$ , i.e. choose to pump the string  $a^M b a^M b$ :

Now  $v$  and  $y$  can't be  $t$ , since  $|vxy| \leq M$ :

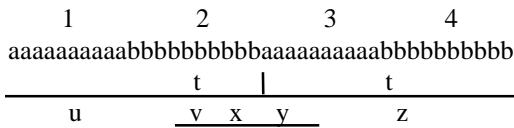


Suppose  $|v| = |y|$ . Now we have to show that repeating them makes the two copies of  $t$  different. But we can't.

$$L = \{tt : t \in \{a, b\}^*\}$$

But let's consider  $L' = L \cap a^*b^*a^*b^*$

This time, we let  $|w| > 2M$ , and the number of both  $a$ 's and  $b$ 's in  $w > M$ :



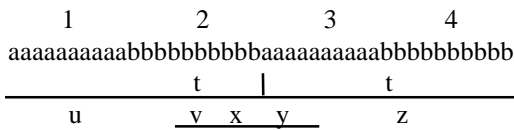
Now we use pumping to show that  $L'$  is not context free.

First, notice that if either  $v$  or  $y$  contains both  $a$ 's and  $b$ 's, then we immediately violate the rules for  $L'$  when we pump.

So now we know that  $v$  and  $y$  must each fall completely in one of the four marked regions.

$$L' = \{tt : t \in \{a, b\}^*\} \cap a^*b^*a^*b^*$$

$|w| > 2M$ , and the number of both  $a$ 's and  $b$ 's in  $w > M$ :



Consider the combinations of  $(v, y)$ :

- (1,1)
- (2,2)
- (3,3)
- (4,4)
- (1,2)
- (2,3)
- (3,4)
- (1,3)
- (2,4)
- (1,4)

## The Context-Free Languages Are Not Closed Under Intersection

Proof: (by counterexample)

Consider  $L = \{a^n b^n c^n : n \geq 0\}$

$L$  is not context-free.

Let  $L_1 = \{a^n b^n c^m : n, m \geq 0\}$  /\* equal a's and b's  
 $L_2 = \{a^m b^n c^n : n, m \geq 0\}$  /\* equal b's and c's

Both  $L_1$  and  $L_2$  are context-free.

But  $L = L_1 \cap L_2$ .

So, if the context-free languages were closed under intersection,  $L$  would have to be context-free. But it isn't.

## The Context-Free Languages Are Not Closed Under Complementation

Proof: (by contradiction)

By definition:

$$\overline{L_1 \cap L_2} = \overline{L_1} \cup \overline{L_2}$$

Since the context-free languages are closed under union, if they were also closed under complementation, they would necessarily be closed under intersection. But we just showed that they are not. Thus they are not closed under complementation.

## The Deterministic Context-Free Languages Are Closed Under Complement

Proof:

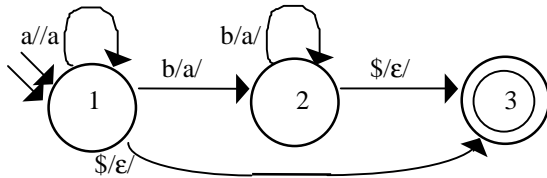
Let  $L$  be a language such that  $L$  is accepted by the deterministic PDA  $M$ . We construct a deterministic PDA  $M'$  to accept (the complement of  $L$ ), just as we did for FSMs:

1. Initially, let  $M' = M$ .
2.  $M'$  is already deterministic.
3. Make  $M'$  simple. Why?
4. Complete  $M'$  by adding a dead state, if necessary, and adding all required transitions into it, including:
  - Transitions that are required to assure that for all input, stack combinations some transition can be followed.
  - If some state  $q$  has a transition on  $(\epsilon, \epsilon)$  and if it does not later lead to a state that does consume something then make a transition on  $(\epsilon, \epsilon)$  to the dead state.
5. Swap final and nonfinal states.
6. Notice that  $M'$  is still deterministic.

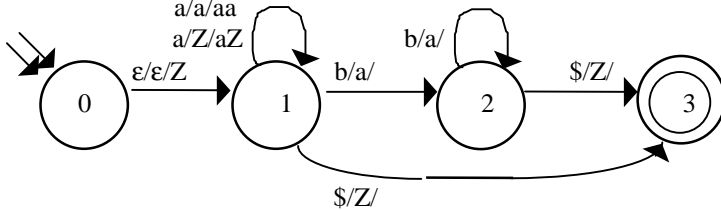


### An Example of the Construction

$L = a^n b^n$  M accepts  $L\$$  (and is deterministic):

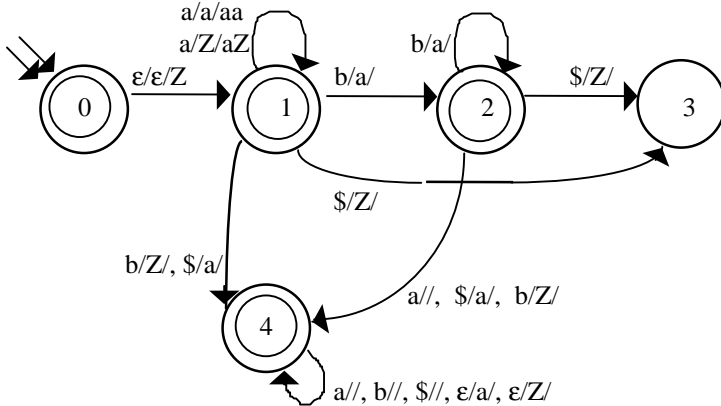


Set  $M = M'$ . Make M simple.



### The Construction, Continued

Add dead state(s) and swap final and nonfinal states:



- Issues:
- 1) Never having the machine die
  - 2)  $\neg(L\$) \neq (\neg L)\$$
  - 3) Keeping the machine deterministic

### Deterministic vs. Nondeterministic Context-Free Languages

**Theorem:** The class of deterministic context-free languages is a *proper* subset of the class of context-free languages.

**Proof:** Consider  $L = \{a^n b^m c^p : m \neq n \text{ or } m \neq p\}$  L is context free (we have shown a grammar for it).

But L is not deterministic. If it were, then its complement  $L_1$  would be deterministic context free, and thus certainly context free. But then

$$L_2 = L_1 \cap a^* b^* c^* \text{ (a regular language)}$$

would be context free. But

$$L_2 = \{a^n b^n c^n : n \geq 0\}, \text{ which we know is not context free.}$$

Thus there exists at least one context-free language that is not deterministic context free.

Note that deterministic context-free languages are **not** closed under union, intersection, or difference.

## Decision Procedures for CFLs & PDAs

### Decision Procedures for CFLs

There are decision procedures for the following ( $G$  is a CFG):

- Deciding whether  $w \in L(G)$ .
- Deciding whether  $L(G) = \emptyset$ .
- Deciding whether  $L(G)$  is finite/infinite.

Such decision procedures usually involve conversions to Chomsky Normal Form or Greibach Normal Form. Why?

**Theorem:** For any context free grammar  $G$ , there exists a number  $n$  such that:

1. If  $L(G) \neq \emptyset$ , then there exists a  $w \in L(G)$  such that  $|w| < n$ .
2. If  $L(G)$  is infinite, then there exists  $w \in L(G)$  such that  $n \leq |w| < 2n$ .

There are **not** decision procedures for the following:

- Deciding whether  $L(G) = \Sigma^*$ .
- Deciding whether  $L(G_1) = L(G_2)$ .

If we could decide these problems, we could decide the halting problem. (More later.)

### Decision Procedures for PDA's

There are decision procedures for the following ( $M$  is a PDA):

- Deciding whether  $w \in L(M)$ .
- Deciding whether  $L(M) = \emptyset$ .
- Deciding whether  $L(M)$  is finite/infinite.

Convert  $M$  to its equivalent CFG and use the corresponding CFG decision procedure. Why avoid using PDA's directly?

There are **not** decision procedures for the following:

- Deciding whether  $L(M) = \Sigma^*$ .
- Deciding whether  $L(M_1) = L(M_2)$ .

If we could decide these problems, we could decide the halting problem. (More later.)

## Comparing Regular and Context-Free Languages

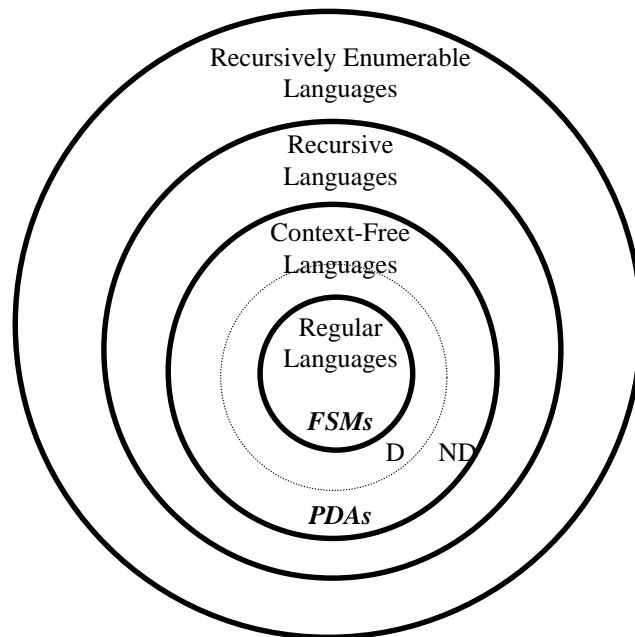
### Regular Languages

- regular exprs.
  - or
- regular grammars
- recognize
- = DFSAs
- recognize
- minimize FSAs
  
- closed under:
  - \* concatenation
  - \* union
  - \* Kleene star
  - \* complement
  - \* intersection
- pumping lemma
- deterministic = nondeterministic

### Context-Free Languages

- context-free grammars
  
- parse
- = NDPDAs
- parse
- find deterministic grammars
- find efficient parsers
- closed under:
  - \* concatenation
  - \* union
  - \* Kleene star
  
- intersection w/ reg. langs
- pumping lemma
- deterministic  $\neq$  nondeterministic

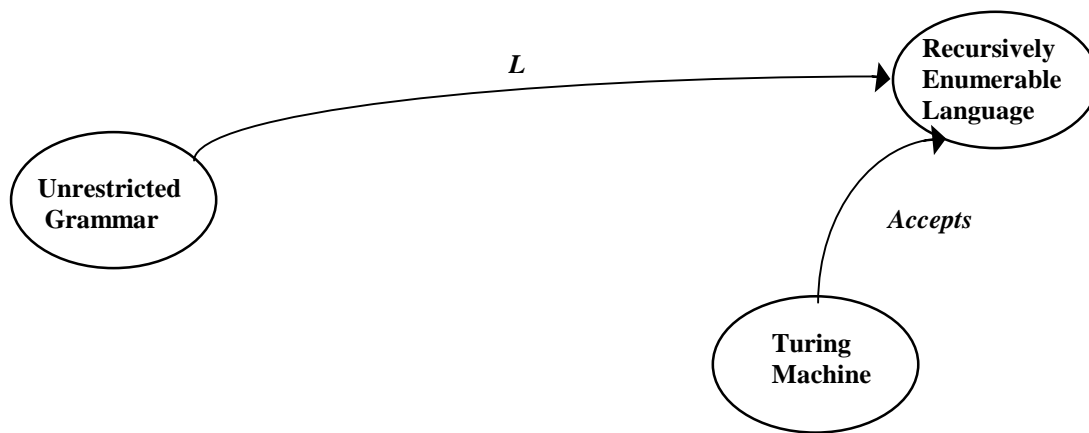
## Languages and Machines



# Turing Machines

Read K & S 4.1.  
Do Homework 17.

## Grammars, Recursively Enumerable Languages, and Turing Machines

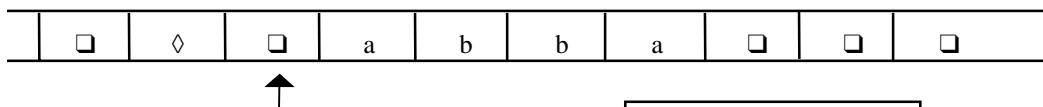


### Turing Machines

Can we come up with a new kind of automaton that has two properties:

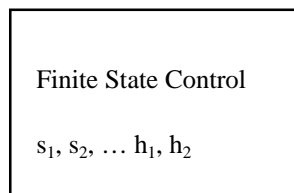
- powerful enough to describe all computable things  
    unlike FSMs and PDAs
- simple enough that we can reason formally about it  
    like FSMs and PDAs  
    unlike real computers

### Turing Machines



At each step, the machine may:

- go to a new state, and
- either
  - write on the current square, or
  - move left or right



### A Formal Definition

A Turing machine is a quintuple  $(K, \Sigma, \delta, s, H)$ :

- $K$  is a finite set of states;
- $\Sigma$  is an alphabet, containing at least  $\square$  and  $\diamond$ , but not  $\rightarrow$  or  $\leftarrow$ ;
- $s \in K$  is the initial state;
- $H \subseteq K$  is the set of halting states;
- $\delta$  is a function from:

$$\begin{array}{ccccccc}
 (K - H) & \times & \Sigma & \text{to} & K & \times & (\Sigma \cup \{\rightarrow, \leftarrow\}) \\
 \text{non-halting state} & \times & \text{input symbol} & & \text{state} & \times & \text{action (write or move)} \\
 & & \text{such that} & & & & 
 \end{array}$$

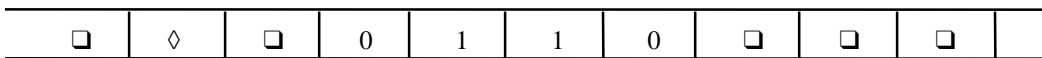
- (a) if the input symbol is  $\diamond$ , the action is  $\rightarrow$ , and
- (b)  $\diamond$  can never be written .

### Notes on the Definition

1. The input tape is infinite to the right (and full of  $\square$ ), but has a wall to the left. Some definitions allow infinite tape in both directions, but it doesn't matter.
2.  $\delta$  is a function, not a relation. So this is a definition for deterministic Turing machines.
3.  $\delta$  must be defined for all state, input pairs unless the state is a halt state.
4. Turing machines do not necessarily halt (unlike FSM's). Why? To halt, they must enter a halt state. Otherwise they loop.
5. Turing machines generate output so they can actually compute functions.

### A Simple Example

A Turing Machine Odd Parity Machine:



$\Sigma = 0, 1, \diamond, \square$

$s =$

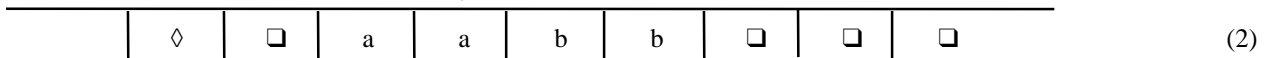
$H =$

$\delta =$

### Formalizing the Operation



(1)



(2)

A **configuration** of a Turing machine

$M = (K, \Sigma, \delta, s, H)$  is a member of

$K$	$\times$	$\diamond\Sigma^*$	$\times$	$(\Sigma^*(\Sigma - \{\square\})) \cup \epsilon$
state		input up to scanned square		input after scanned square

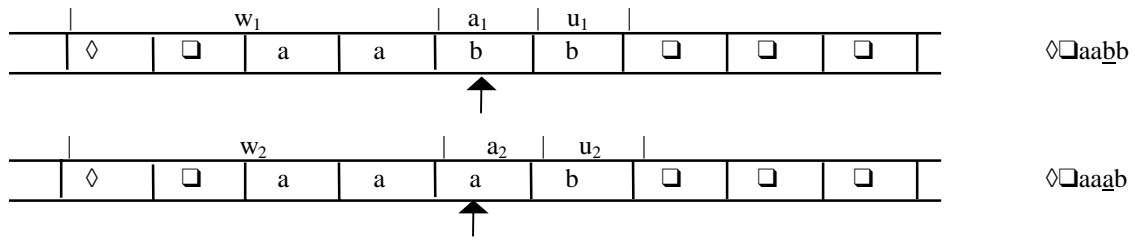
The input after the scanned square may be empty, but it may not end with a blank. We assume the entire tape to the right of the input is filled with blanks.

- (1)  $(q, \diamond aab, b) = (q, \diamond aabb)$   
 (2)  $(h, \diamond \square aabb, \epsilon) = (h, \diamond \square aabb)$  a halting configuration

### Yields

$(q_1, w_1 \underline{a_1} u_1) \vdash_M (q_2, w_2 \underline{a_2} u_2)$ ,  $a_1$  and  $a_2 \in \Sigma$ , iff  $\exists b \in \Sigma \cup \{\leftarrow, \rightarrow\}, \delta(q_1, a_1) = (q_2, b)$  and either:

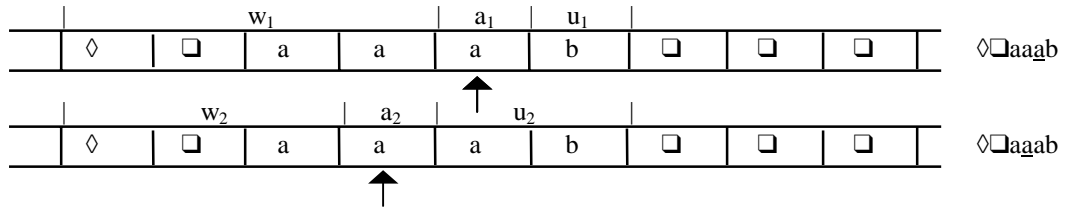
(1)  $b \in \Sigma, w_1 = w_2, u_1 = u_2$ , and  $a_2 = b$  (rewrite without moving the head)



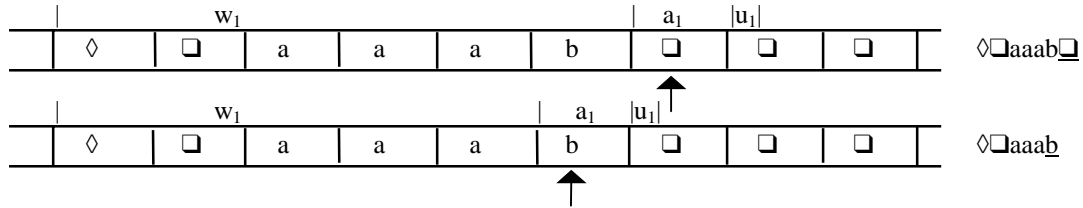
### Yields, Continued

(2)  $b = \leftarrow, w_1 = w_2 a_2$ , and either

(a)  $u_2 = a_1 u_1$ , if  $a_1 \neq \square$  or  $u_1 \neq \epsilon$ ,



or (b)  $u_2 = \epsilon$ , if  $a_1 = \square$  and  $u_1 = \epsilon$

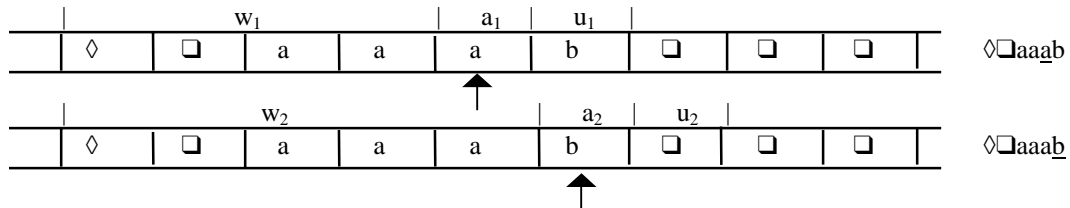


If we scan left off the first square of the blank region, then drop that square from the configuration.

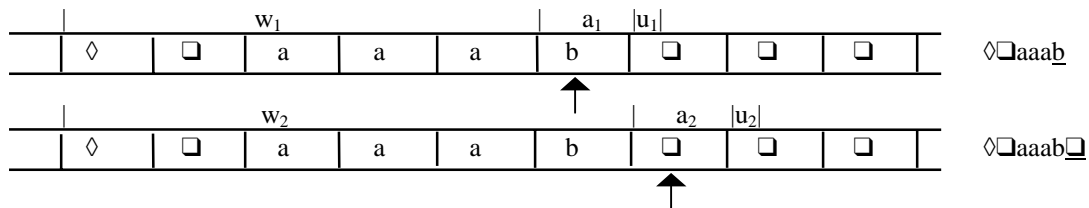
### Yields, Continued

(3)  $b = \rightarrow, w_2 = w_1 a_1$ , and either

(a)  $u_1 = a_2 u_2$



or (b)  $u_1 = u_2 = \epsilon$  and  $a_2 = \square$



If we scan right onto the first square of the blank region, then a new blank appears in the configuration.

## Yields, Continued

For any Turing machine  $M$ , let  $\vdash_M^*$  be the reflexive, transitive closure of  $\vdash_M$ .

Configuration  $C_1$  **yields** configuration  $C_2$  if

$$C_1 \vdash_M^* C_2.$$

A **computation** by  $M$  is a sequence of configurations  $C_0, C_1, \dots, C_n$  for some  $n \geq 0$  such that

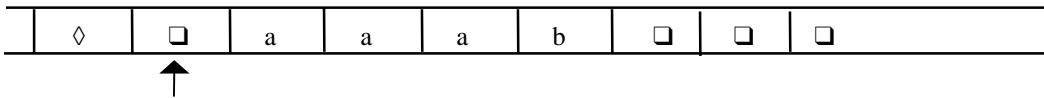
$$C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \dots \vdash_M C_n.$$

We say that the computation is of **length**  $n$  or that it has  $n$  **steps**, and we write

$$C_0 \vdash_M^n C_n$$

### A Context-Free Example

$M$  takes a tape of a's then b's, possibly with more a's, and adds b's as required to make the number of b's equal the number of a's.



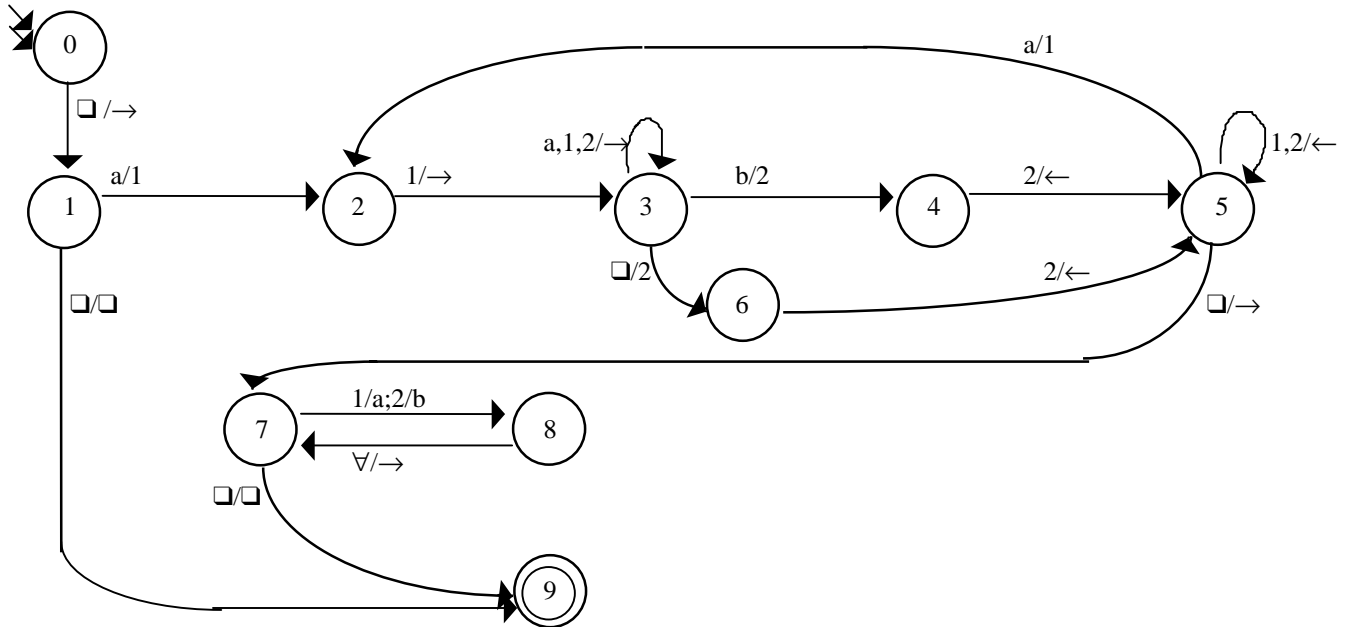
$K = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

$\Sigma = a, b, \diamond, \square, \uparrow, \downarrow$

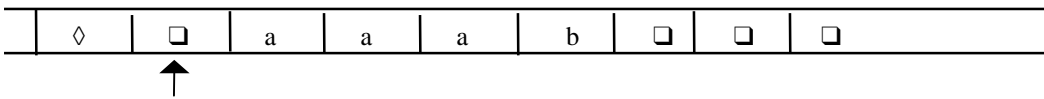
$s = 0$

$H = \{9\}$

$\delta =$



**An Example Computation**



- $(0, \diamond \square \underline{a} a a b) \vdash_M$
- $(1, \diamond \square \underline{a} a a b) \vdash_M$
- $(2, \diamond \square \underline{1} a a b) \vdash_M$
- $(3, \diamond \square \underline{1} \underline{a} a b) \vdash_M$
- $(3, \diamond \square \underline{1} a \underline{a} b) \vdash_M$
- $(3, \diamond \square \underline{1} a a \underline{b}) \vdash_M$
- $(4, \diamond \square \underline{1} a a \underline{2}) \vdash_M$

...

## Notes on Programming

The machine has a strong procedural feel.

It's very common to have state pairs, in which the first writes on the tape and the second moves. Some definitions allow both actions at once, and those machines will have fewer states.

There are common idioms, like scan left until you find a blank.

Even a very simple machine is a nuisance to write.

### A Notation for Turing Machines

(1) Define some basic machines

- Symbol writing machines

For each  $a \in \Sigma - \{\diamond\}$ , define  $M_a$ , written just  $a$ ,  $= (\{s, h\}, \Sigma, \delta, s, \{h\})$ ,

for each  $b \in \Sigma - \{\diamond\}$ ,  $\delta(s, b) = (h, a)$

$\delta(s, \diamond) = (s, \rightarrow)$

Example:

$a$  writes an  $a$

- Head moving machines

For each  $a \in \{\leftarrow, \rightarrow\}$ , define  $M_a$ , written  $R(\rightarrow)$  and  $L(\leftarrow)$ :

for each  $b \in \Sigma - \{\diamond\}$ ,  $\delta(s, b) = (h, a)$

$\delta(s, \diamond) = (s, \rightarrow)$

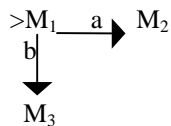
Examples:

$R$  moves one square to the right

$aR$  writes an  $a$  and then moves one square to the right.

### A Notation for Turing Machines, Cont'd

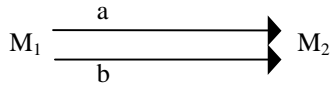
(2) The rules for combining machines: as with FSMs



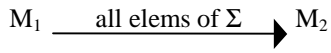
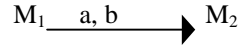
- Start in the start state of  $M_1$ .
- Compute until  $M_1$  reaches a halt state.
- Examine the tape and take the appropriate transition.
- Start in the start state of the next machine, etc.
- Halt if any component reaches a halt state and has no place to go.
- If any component fails to halt, then the entire machine may fail to halt.



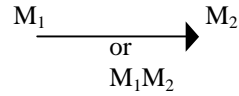
### Shorthands



becomes



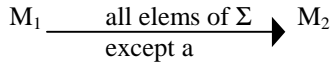
becomes



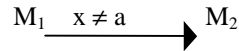
MM

becomes

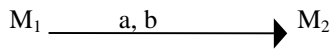
$M^2$



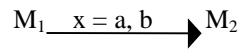
becomes



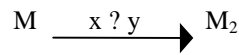
and x takes on the value of the current square



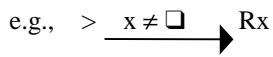
becomes



and x takes on the value of the current square

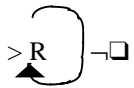


if  $x = y$  then take the transition



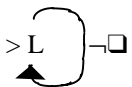
if the current square is not blank, go right and copy it.

### Some Useful Machines



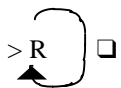
find the first blank square to the right of the current square

$R_{\square}$



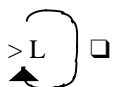
find the first blank square to the left of the current square

$L_{\square}$



find the first nonblank square to the right of the current square

$R_{\neq \square}$



find the first nonblank square to the left of the current square

$L_{\neq \square}$

### More Useful Machines

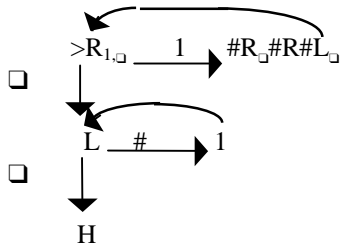
- $L_a$  find the first occurrence of a to the left of the current square
- $R_{a,b}$  find the first occurrence of a or b to the right of the current square
- $L_{a,b} \xrightarrow{a} M_1$  find the first occurrence of a or b to the left of the current square, then go to  $M_1$  if the detected character is a; go to  $M_2$  if the detected character is b  
 $\downarrow$   
 $M_2$
- $L_{x=a,b}$  find the first occurrence of a or b to the left of the current square and set x to the value found
- $L_{x=a,b}Rx$  find the first occurrence of a or b to the left of the current square, set x to the value found, move one square to the right, and write x (a or b)

### An Example

Input:  $\diamond \square w$   $w \in \{1\}^*$

Output:  $\diamond \square w^3$

Example:  $\diamond \square 111 \square \square \square \square \square \square \square \square \square \square \square \square \square \square \square \square$

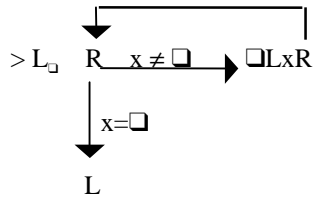


### A Shifting Machine $S_{\leftarrow}$

Input:  $\square \square w \square$

Output:  $\square w \square$

Example:  $\square \square abba \square \square \square \square \square \square \square \square \square \square \square \square \square \square$



# Computing with Turing Machines

Read K & S 4.2.  
Do Homework 18.

## Turing Machines as Language Recognizers

Convention: We will write the input on the tape as:

$$\diamond \square w \square, w \text{ contains no } \square\text{s}$$

The initial configuration of M will then be:

$$(s, \diamond \square w)$$

A recognizing Turing machine M must have two halting states: y and n

Any configuration of M whose state is:

y is an accepting configuration

n is a rejecting configuration

Let  $\Sigma_0$ , the input alphabet, be a subset of  $\Sigma_M - \{\square, \diamond\}$

Then M **decides** a language  $L \subseteq \Sigma_0^*$  iff for any string

$w \in \Sigma_0^*$  it is true that:

if  $w \in L$  then M accepts w, and

if  $w \notin L$  then M rejects w.

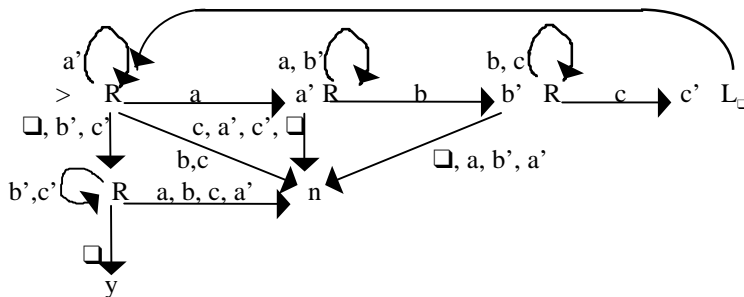
A language L is **recursive** if there is a Turing machine M that decides it.

### A Recognition Example

$$L = \{a^n b^n c^n : n \geq 0\}$$

Example:  $\diamond \square aabbcc \square \square \square \square \square \square \square \square$

Example:  $\diamond \square aaccb \square \square \square \square \square \square \square \square$

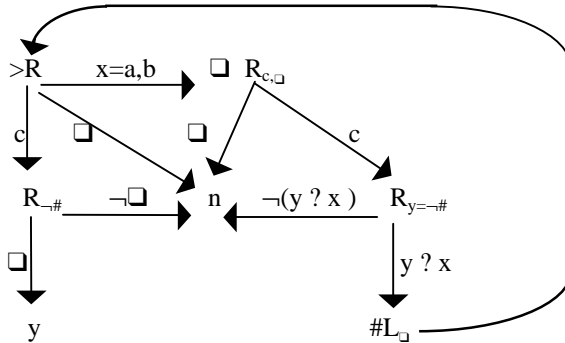


### Another Recognition Example

$$L = \{wcw : w \in \{a, b\}^*\}$$

Example:  $\diamond \square abbcabb \square \square \square$

Example:  $\diamond \square acabb \square \square \square$



### Do Turing Machines Stop?

FSMs Always halt after  $n$  steps, where  $n$  is the length of the input. At that point, they either accept or reject.

PDAs Don't always halt, but there is an algorithm to convert any PDA into one that does halt.

Turing machines Can do one of three things:

- (1) Halt and accept
- (2) Halt and reject
- (3) Not halt

And now there is no algorithm to determine whether a given machine always halts.

### Computing Functions

Let  $\Sigma_0 \subseteq \Sigma - \{\diamond, \square\}$  and let  $w \in \Sigma_0^*$

Convention: We will write the input on the tape as:  $\diamond \square w \square$

The initial configuration of  $M$  will then be:  $(s, \diamond \square w)$

Define  $M(w) = y$  iff:

- $M$  halts if started in the input configuration,
- the tape of  $M$  when it halts is  $\diamond \square y \square$ , and
- $y \in \Sigma_0^*$

Let  $f$  be any function from  $\Sigma_0^*$  to  $\Sigma_0^*$ .

We say that  $M$  **computes**  $f$  if, for all  $w \in \Sigma_0^*$ ,  $M(w) = f(w)$

A function  $f$  is **recursive** if there is a Turing machine  $M$  that computes it.

### Example of Computing a Function

$$f(w) = ww$$

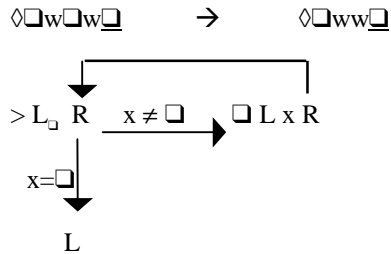
Input:  $\diamond \square w \square \square \square \square \square \square$

Output:  $\diamond \square ww \square$

Define the copy machine C:

$\diamond \square w \square \square \square \square \square \square \rightarrow \diamond \square w \square w \square$

Remember the  $S_{\leftarrow}$  machine:



Then the machine to compute f is just  $>C S L_{\leftarrow}$

### Computing Numeric Functions

We say that a Turing machine M computes a function f from  $N^k$  to N provided that

$$\text{num}(M(n_1; n_2; \dots n_k)) = f(\text{num}(n_1), \dots \text{num}(n_k))$$

Example:  $\text{Succ}(n) = n + 1$

We will represent n in binary. So  $n \in 0 \cup 1\{0,1\}^*$

Input:  $\diamond \square n \square \square \square \square \square \square$

Output:  $\diamond \square n+1 \square$

$\diamond \square 1111 \square \square \square \square$

Output:  $\diamond \square 10000 \square$

### Why Are We Working with Our Hands Tied Behind Our Backs?

Turing machines are more powerful than any of the other formalisms we have studied so far.

Turing machines are a **lot** harder to work with than all the real computers we have available.

Why bother?

The very simplicity that makes it hard to program Turing machines makes it possible to reason formally about what they can do. If we can, once, show that anything a real computer can do can be done (albeit clumsily) on a Turing machine, then we have a way to reason about what real computers can do.

# Recursively Enumerable and Recursive Languages

Read K & S 4.5.

## Recursively Enumerable Languages

Let  $\Sigma_0$ , the input alphabet to a Turing machine  $M$ , be a subset of  $\Sigma_M - \{\square, \diamond\}$

Let  $L \subseteq \Sigma_0^*$ .

$M$  semidecides  $L$  iff

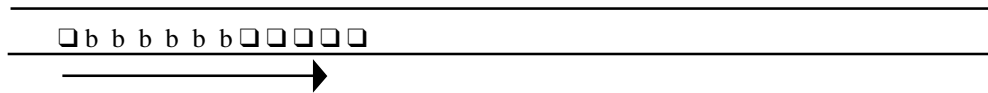
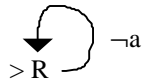
for any string  $w \in \Sigma_0^*$ ,

$w \in L \Rightarrow$   $M$  halts on input  $w$   
 $w \notin L \Rightarrow$   $M$  does not halt on input  $w$   
 $M(w) = \uparrow$

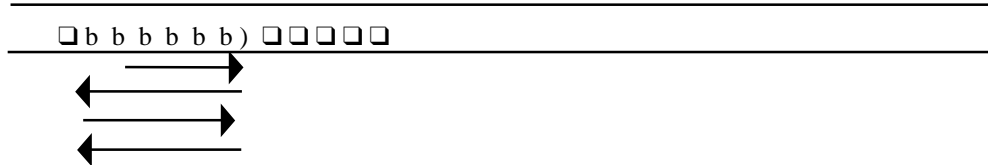
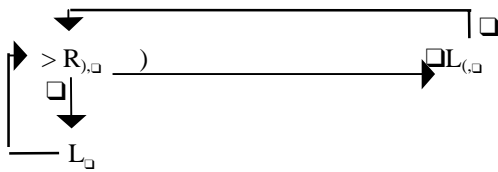
$L$  is **recursively enumerable** iff there is a Turing machine that semidecides it.

## Examples of Recursively Enumerable Languages

$L = \{w \in \{a, b\}^* : w \text{ contains at least one } a\}$



$L = \{w \in \{a, b, (, )\}^* : w \text{ contains at least one set of balanced parentheses}\}$



## Recursively Enumerable Languages that Aren't Also Recursive

**A Real Life Example:**

$L = \{w \in \{\text{friends}\} : w \text{ will answer the message you've just sent out}\}$

**Theoretical Examples**

$L = \{\text{Turing machines that halt on a blank input tape}\}$

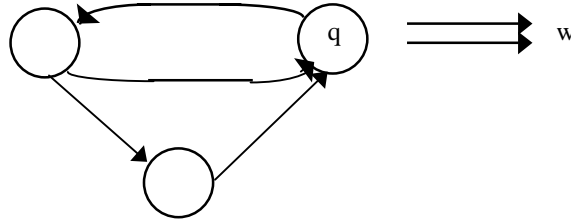
Theorems with valid proofs.

## Why Are They Called Recursively Enumerable Languages?

Enumerate means list.

We say that Turing machine  $M$  **enumerates** the language  $L$  iff, for some fixed state  $q$  of  $M$ ,

$$L = \{w : (s, \diamond \square) \vdash_M^* (q, \diamond \square w)\}$$



A language is **Turing-enumerable** iff there is a Turing machine that enumerates it.

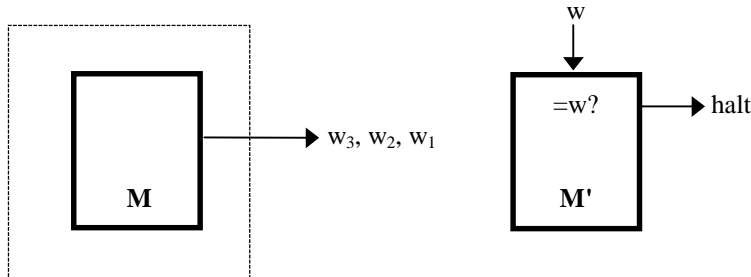
Note that  $q$  is not a halting state. It merely signals that the current contents of the tape should be viewed as a member of  $L$ .

### Recursively Enumerable and Turing Enumerable

**Theorem:** A language is recursively enumerable iff it is Turing-enumerable.

**Proof** that Turing-enumerable implies RE: Let  $M$  be the Turing machine that enumerates  $L$ . We convert  $M$  to a machine  $M'$  that semidecides  $L$ :

1. Save input  $w$ .
2. Begin enumerating  $L$ . Each time an element of  $L$  is enumerated, compare it to  $w$ . If they match, accept.



### The Other Way

**Proof** that RE implies Turing-enumerable:

If  $L \subseteq \Sigma^*$  is a recursively enumerable language, then there is a Turing machine  $M$  that semidecides  $L$ .

A procedure to enumerate all elements of  $L$ :

Enumerate all  $w \in \Sigma^*$  lexicographically.

e.g.,  $\epsilon$ ,  $a$ ,  $b$ ,  $aa$ ,  $ab$ ,  $ba$ ,  $bb$ , ...

As each string  $w_i$  is enumerated:

1. Start up a copy of  $M$  with  $w_i$  as its input.
2. Execute one step of each  $M_i$  initiated so far, excluding only those that have previously halted.
3. Whenever an  $M_i$  halts, output  $w_i$ .

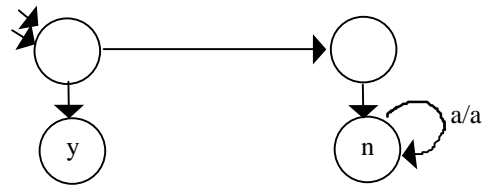
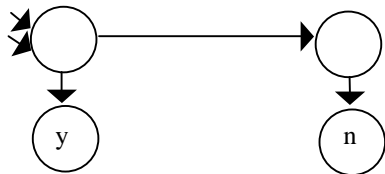
$\epsilon$ [1]					
$\epsilon$ [2]	$a$ [1]				
$\epsilon$ [3]	$a$ [2]	$b$ [1]			
$\epsilon$ [4]	$a$ [3]	$b$ [2]	$aa$ [1]		
$\epsilon$ [5]	$a$ [4]	<u><math>b</math></u> [3]	$aa$ [2]	$ab$ [1]	
$\epsilon$ [6]	$a$ [5]		$aa$ [3]	$ab$ [2]	$ba$ [1]

## Every Recursive Language is Recursively Enumerable

If  $L$  is recursive, then there is a Turing machine that decides it.

From  $M$ , we can build a new Turing machine  $M'$  that semidecides  $L$ :

1. Let  $n$  be the reject (and halt) state of  $M$ .
2. Then add to  $\delta'$   
 $((n, a), (n, a))$  for all  $a \in \Sigma$



What about the other way around?

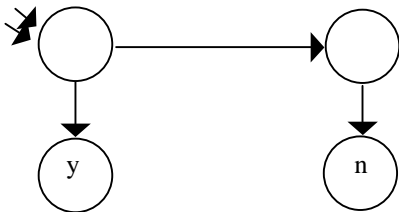
Not true. There are recursively enumerable languages that are not recursive.

## The Recursive Languages Are Closed Under Complement

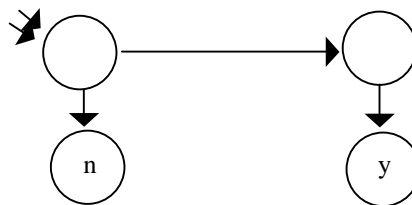
Proof: (by construction) If  $L$  is recursive, then there is a Turing machine  $M$  that decides  $L$ .

We construct a machine  $M'$  to decide  $\bar{L}$  by taking  $M$  and swapping the roles of the two halting states  $y$  and  $n$ .

$M$ :



$M'$ :

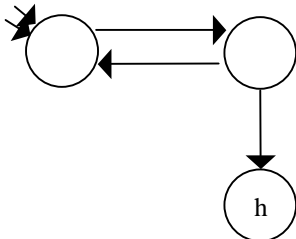


This works because, by definition,  $M$  is

- deterministic
- complete

## Are the Recursively Enumerable Languages Closed Under Complement?

$M$ :



$M'$ :

**Lemma:** There exists at least one language  $L$  that is recursively enumerable but not recursive.

**Proof** that  $M'$  doesn't exist: Suppose that the RE languages were closed under complement. Then if  $L$  is RE,  $\bar{L}$  would be RE. If that were true, then  $\bar{L}$  would also be recursive because we could construct  $M$  to decide it:

1. Let  $T_1$  be the Turing machine that semidecides  $L$ .
2. Let  $T_2$  be the Turing machine that semidecides  $\bar{L}$ .
3. Given a string  $w$ , fire up both  $T_1$  and  $T_2$  on  $w$ . Since any string in  $\Sigma^*$  must be in either  $L$  or  $\bar{L}$ , one of the two machines will eventually halt. If it's  $T_1$ , accept; if it's  $T_2$ , reject.

But we know that there is at least one RE language that is not recursive. Contradiction.



## Recursive and RE Languages

**Theorem:** A language is recursive iff both it and its complement are recursively enumerable.

**Proof:**

- L recursive implies L and  $\neg L$  are RE: Clearly L is RE. And, since the recursive languages are closed under complement,  $\neg L$  is recursive and thus also RE.
- L and  $\neg L$  are RE implies L recursive: Suppose L is semidecided by M1 and  $\neg L$  is semidecided by M2. We construct M to decide L by using two tapes and simultaneously executing M1 and M2. One (but not both) must eventually halt. If it's M1, we accept; if it's M2 we reject.

### Lexicographic Enumeration

We say that M **lexicographically enumerates** L if M enumerates the elements of L in lexicographic order. A language L is **lexicographically Turing-enumerable** iff there is a Turing machine that lexicographically enumerates it.

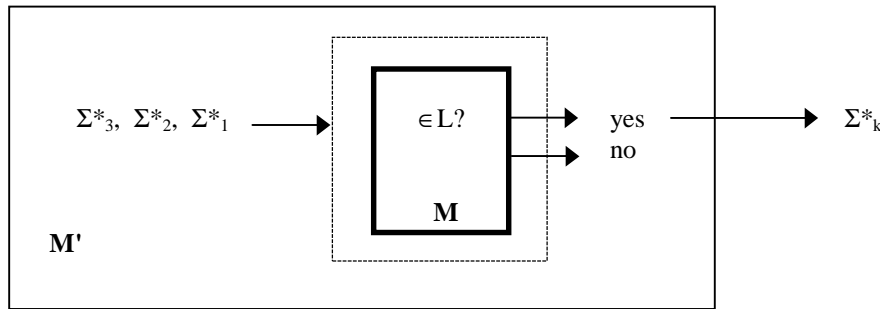
Example:  $L = \{a^n b^n c^n\}$

Lexicographic enumeration:

### Proof

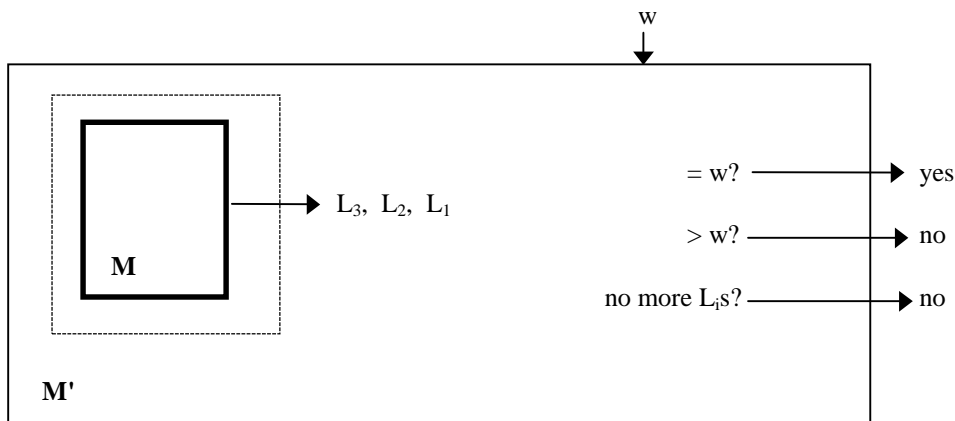
**Theorem:** A language is recursive iff it is lexicographically Turing-enumerable.

**Proof** that recursive implies lexicographically Turing enumerable: Let M be a Turing machine that decides L. Then  $M'$  lexicographically generates the strings in  $\Sigma^*$  and tests each using M. It outputs those that are accepted by M. Thus  $M'$  lexicographically enumerates L.



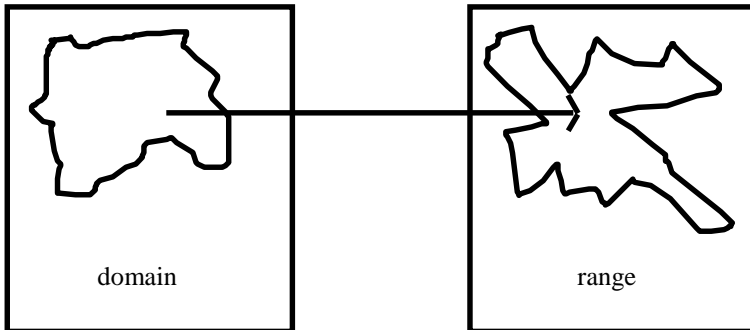
### Proof, Continued

**Proof** that lexicographically Turing enumerable implies recursive: Let M be a Turing machine that lexicographically enumerates L. Then, on input w,  $M'$  starts up M and waits until either M generates w (so  $M'$  accepts), M generates a string that comes after w (so  $M'$  rejects), or M halts (so  $M'$  rejects). Thus  $M'$  decides L.



## Partially Recursive Functions

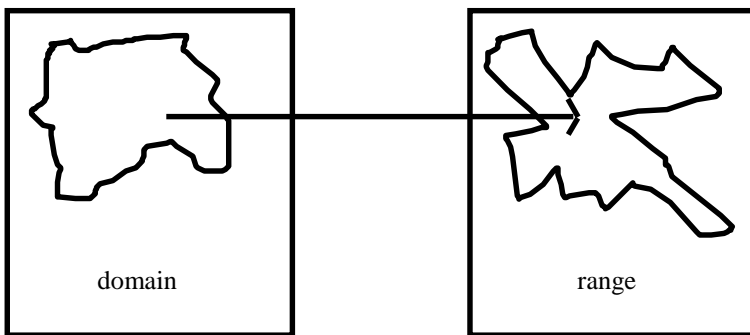
	Languages	Functions
Tm always halts	<b>recursive</b>	<b>recursive</b>
Tm halts if yes	<b>recursively enumerable</b>	?



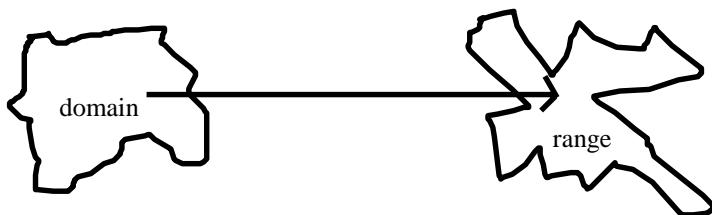
Suppose we have a function that is not defined for all elements of its domain.

Example:  $f: \mathbb{N} \rightarrow \mathbb{N}, f(n) = n/2$

## Partially Recursive Functions

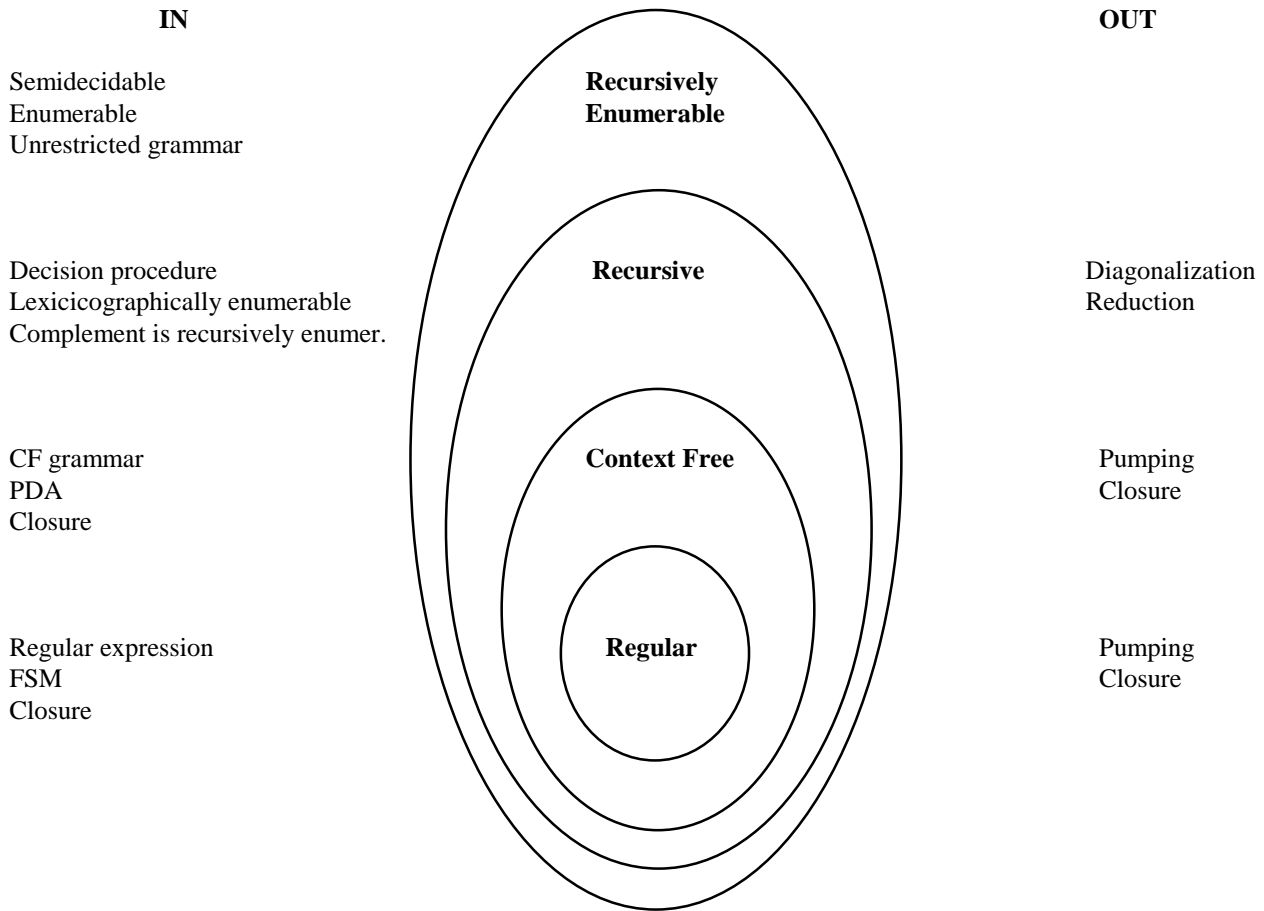


One solution: Redefine the domain to be exactly those elements for which  $f$  is defined:



But what if we don't know? What if the domain is not a recursive set (but it is recursively enumerable)? Then we want to define the domain as some larger, recursive set and say that the function is partially recursive. There exists a Turing machine that halts if given an element of the domain but does not halt otherwise.

# Language Summary



# Turing Machine Extensions

Read K & S 4.3.1, 4.4.  
Do Homework 19.

## Turing Machine Definitions

An alternative definition of a Turing machine:

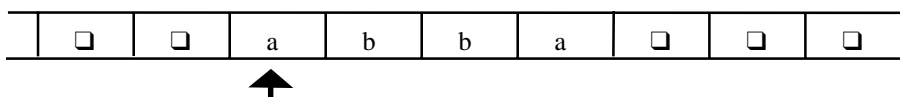
$(K, \Sigma, \Gamma, \delta, s, H)$ :

$\Gamma$  is a finite set of allowable tape symbols. One of these is  $\square$ .

$\Sigma$  is a subset of  $\Gamma$  not including  $\square$ , the input symbols.

$\delta$  is a function from:

$K \times \Gamma$  to  $K \times (\Gamma - \{\square\}) \times \{\leftarrow, \rightarrow\}$   
state, tape symbol, L or R



Example transition:  $((s, a), (s, b, \rightarrow))$

## Do these Differences Matter?

Remember the goal:

Define a device that is:

- powerful enough to describe all computable things,
- simple enough that we can reason formally about it

Both definitions are simple enough to work with, although details may make specific arguments easier or harder.

But, do they differ in their power?

Answer: No.

Consider the differences:

- One way or two way infinite tape: we're about to show that we can simulate two way infinite with ours.
- Rewrite and move at the same time: just affects (linearly) the number of moves it takes to solve a problem.

## Turing Machine Extensions

In fact, there are lots of extensions we can make to our basic Turing machine model. They may make it easier to write Turing machine programs, but none of them increase the power of the Turing machine because:

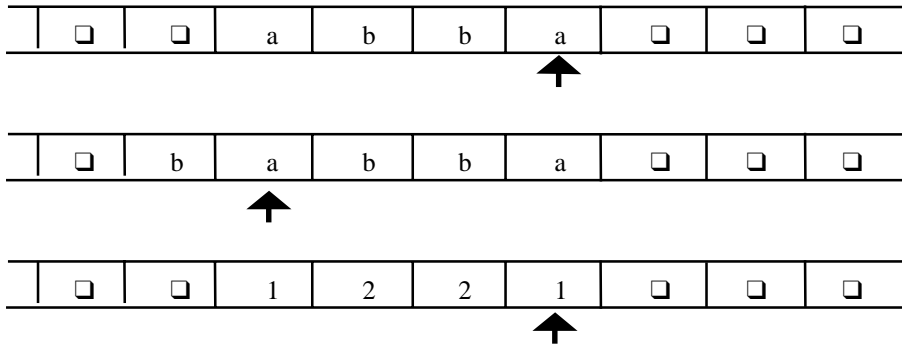
**We can show that every extended machine has an equivalent basic machine.**

We can also place a bound on any change in the complexity of a solution when we go from an extended machine to a basic machine.

Some possible extensions:

- Multiple tapes
- Two-way infinite tape
- Multiple read heads
- Two dimensional "sheet" instead of a tape
- Random access machine
- Nondeterministic machine

## Multiple Tapes



The transition function for a k-tape Turing machine:

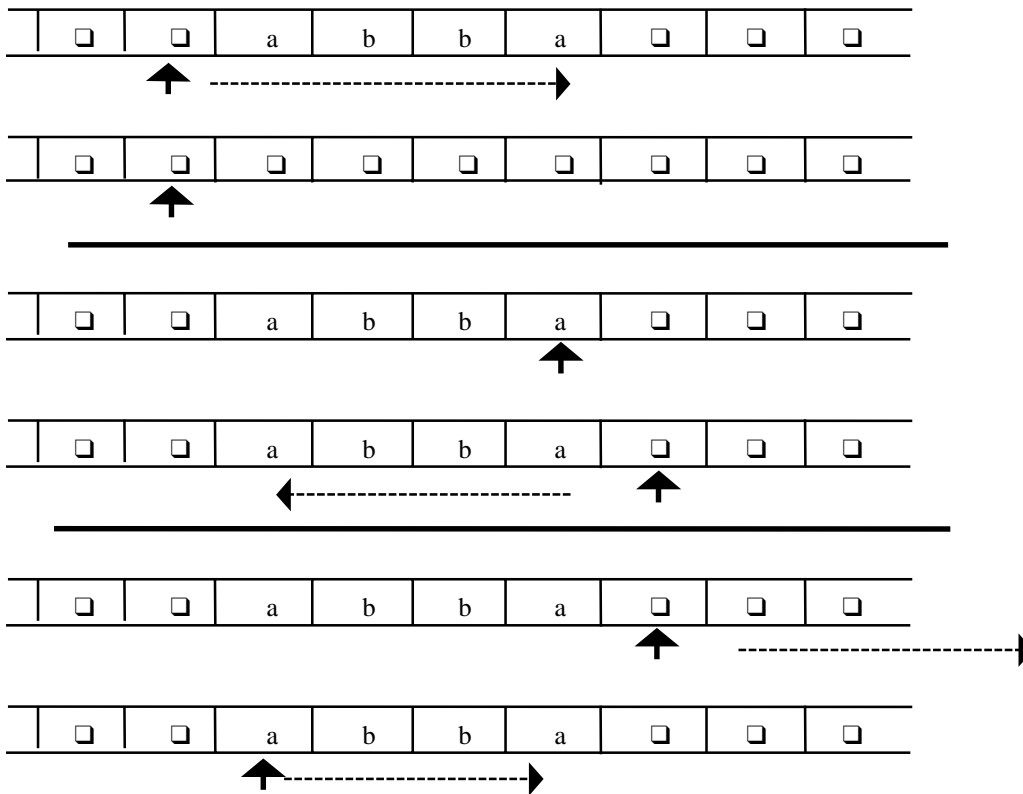
$((K-H), \Sigma_1, \dots, \Sigma_k)$  to  $(K, \Sigma_1 \cup \{\leftarrow, \rightarrow\}, \Sigma_2 \cup \{\leftarrow, \rightarrow\}, \dots, \Sigma_k \cup \{\leftarrow, \rightarrow\})$

Input: input as before on tape 1, others blank

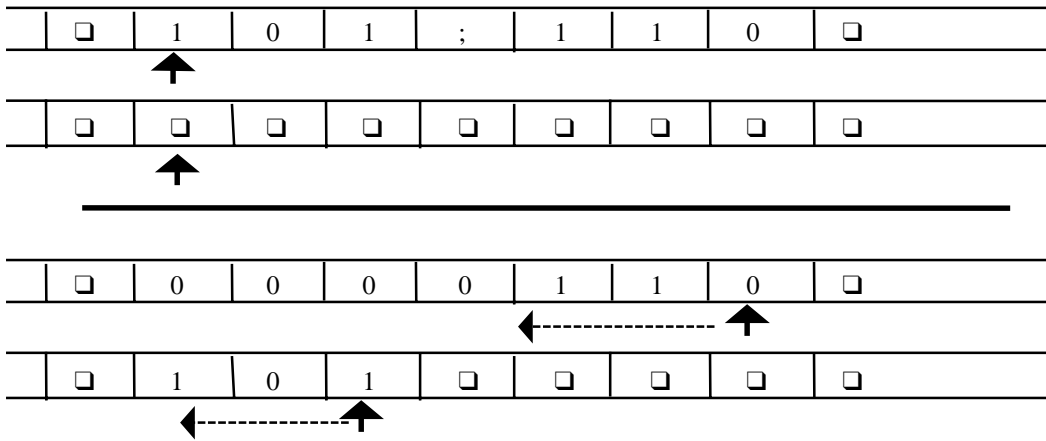
Output: output as before on tape 1, others ignored

### An Example of a Two Tape Machine

Copying a string



### Another Two Tape Example - Addition



### Adding Tapes Adds No Power

**Theorem:** Let  $M$  be a  $k$ -tape Turing machine for some  $k \geq 1$ . Then there is a standard Turing machine  $M'$  where  $\Sigma \subseteq \Sigma'$ , and such that:

- For any input string  $x$ ,  $M$  on input  $x$  halts with output  $y$  on the first tape iff  $M'$  on input  $x$  halts at the same halting state and with the same output on its tape.
- If, on input  $x$ ,  $M$  halts after  $t$  steps, then  $M'$  halts after a number of steps which is  $O(t \cdot (|x| + t))$ .

**Proof:** By construction

$\diamond$	$\diamond$	$\square$	a	b	a	$\square$	$\square$	$\square$	$\square$
	0	0	1	0	0	0	0		
	$\diamond$	a	b	b	a	b	a		
	0	1	0	0	0	0	0		

Alphabet ( $\Sigma'$ ) of  $M' = \Sigma \cup (\Sigma \times \{0, 1\})^k$   
 e.g.,  $\diamond, (\diamond, 0, \diamond, 0), (\square, 0, a, 1)$

### The Operation of $M'$

$\diamond$	$\diamond$	$\square$	a	b	a	$\square$	$\square$	$\square$	$\square$
	0	0	1	0	0	0	0		
	$\diamond$	a	b	b	a	b	a		
	0	1	0	0	0	0	0		

1. Set up the multitrack tape:
  - 1) Shift input one square to right, then set up each square appropriately.
2. Simulate the computation of  $M$  until (if)  $M$  would halt: (start each step to the right of the divided tape)
  - 1) Scan left and store in the state the  $k$ -tuple of characters under the read heads. Move back right.
  - 2) Scan left and update each track as required by the transitions of  $M$ . Move back right.
    - i) If necessary, subdivide a new square into tracks.
3. When  $M$  would halt, reformat the tape to throw away all but track 1, position the head correctly, then go to  $M$ 's halt state.

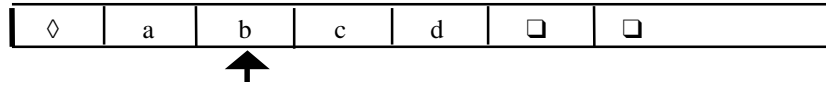
### How Many Steps Does $M'$ Take?

Let:  $x$  be the input string, and  
 $t$  be the number of steps it takes  $M$  to execute.

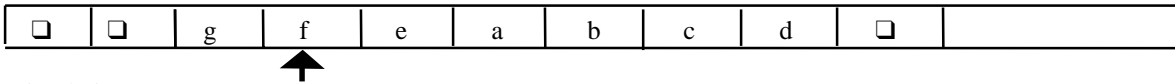
Step 1 (initialization)  $O(|x|)$   
 Step 2 (computation)  
 Number of passes =  $t$   
 Work at each pass:  
 $2.1 = 2 \cdot (\text{length of tape})$   
 $= 2 \cdot (|x| + 2 + t)$   
 $2.2 = 2 \cdot (|x| + 2 + t)$   
 Total =  $O(t \cdot (|x| + t))$   
 Step 3 (clean up)  $O(\text{length of tape})$   
 Total =  $O(t \cdot (|x| + t))$

## Two-Way Infinite Tape

Our current definition:

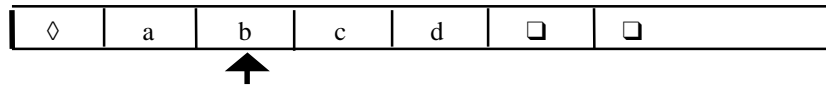


Proposed definition:



Simulation:

Track 1



Track 2



## Simulating a PDA

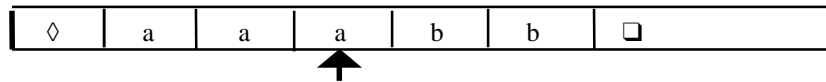
The components of a PDA:

- Finite state controller
- Input tape
- Stack

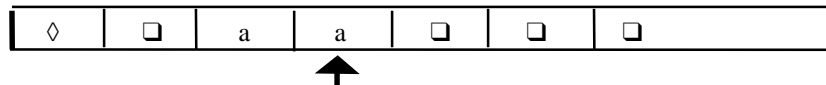
The simulation:

- Finite state controller:
- Input tape:
- Stack:

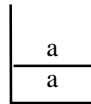
Track 1  
(Input)



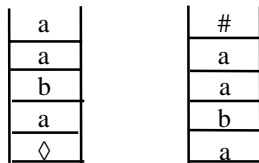
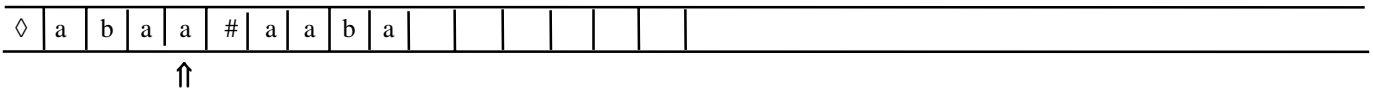
Track 2



Corresponding to



## Simulating a Turing Machine with a PDA with Two Stacks



## Random Access Turing Machines

A random access Turing machine has:

- a fixed number of registers
- a finite length program, composed of instructions with operators such as read, write, load, store, add, sub, jump
- a tape
- a program counter

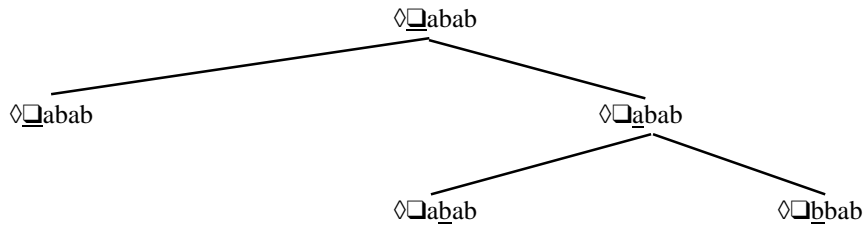
**Theorem:** Standard Turing machines and random access Turing machines compute the same things. Furthermore, the number of steps it takes a standard machine is bounded by a polynomial in the number of steps it takes a random access machine.

## Nondeterministic Turing Machines

A **nondeterministic** Turing machine is a quintuple  $(K, \Sigma, \Delta, s, H)$

where  $K, \Sigma, s,$  and  $H$  are as for standard Turing machines, and  $\Delta$  is a *subset* of

$$((K - H) \times \Sigma) \times (K \times (\Sigma \cup \{\leftarrow, \rightarrow\}))$$



What does it mean for a nondeterministic Turing machine to compute something?

- Semidecides - at least one halts.
- Decides - ?
- Computes - ?

## Nondeterministic Semideciding

Let  $M = (K, \Sigma, \Delta, s, H)$  be a nondeterministic Turing machine. We say that  $M$  **accepts** an input

$$w \in (\Sigma - \{\diamond, \square\})^* \text{ iff}$$

$(s, \diamond \square w)$  yields a least one accepting configuration.

We say that  $M$  **semidecides** a language

$$L \subseteq (\Sigma - \{\diamond, \square\})^* \text{ iff}$$

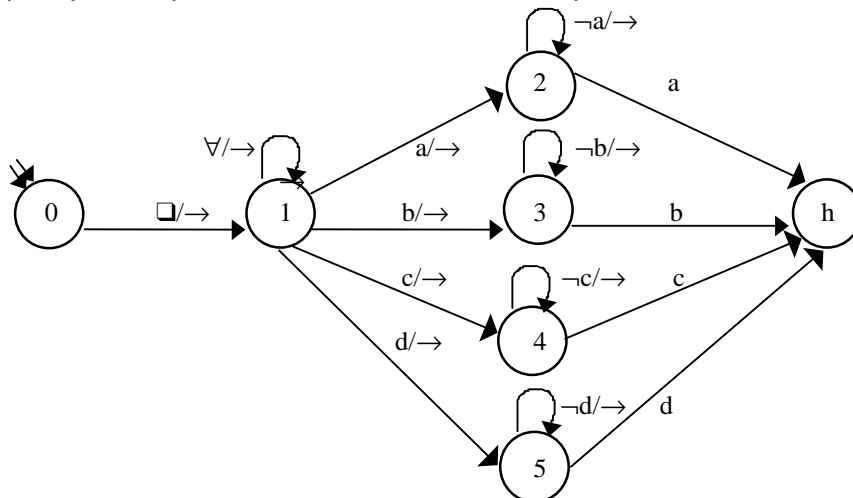
for all  $w \in (\Sigma - \{\diamond, \square\})^*$ :

$$w \in L \text{ iff}$$

$(s, \diamond \square w)$  yields a least one halting configuration.

## An Example

$L = \{w \in \{a, b, c, d\}^* : \text{there are two of at least one letter}\}$





## Nondeterministic Deciding and Computing

M **decides** a language L if, for all  $w \in (\Sigma - \{\diamond, \square\})^*$  :

1. all of M's computations on w halt, and
2.  $w \in L$  iff *at least one* of M's computations accepts.

M **computes** a function f if, for all  $w \in (\Sigma - \{\diamond, \square\})^*$  :

1. all of M's computations halt, and
2. *all* of M's computations result in f(w)

Note that all of M's computations halt iff:

There is a natural number N, depending on M and w, such that there is no configuration C satisfying  $(s, \diamond \square w) \vdash_M^N C$ .

### An Example of Nondeterministic Deciding

$L = \{w \in \{0, 1\}^* : w \text{ is the binary encoding of a composite number}\}$

M decides L by doing the following on input w:

1. Nondeterministically choose two binary numbers  $1 < p, q$ , where  $|p|$  and  $|q| \leq |w|$ , and write them on the tape, after w, separated by ;.

$\diamond \square 110011;111;1111 \square \square$

2. Multiply p and q and put the answer, A, on the tape, in place of p and q.

$\diamond \square 110011;1011111 \square \square$

3. Compare A and w. If equal, go to y. Else go to n.

### Equivalence of Deterministic and Nondeterministic Turing Machines

**Theorem:** If a nondeterministic Turing machine M semidecides or decides a language, or computes a function, then there is a standard Turing machine M' semideciding or deciding the same language or computing the same function.

Note that while nondeterminism doesn't change the computational power of a Turing Machine, it can exponentially increase its speed!

**Proof:** (by construction)

For semideciding: We build M', which runs through all possible computations of M. If one of them halts, M' halts

Recall the way we did this for FSMs: simulate being in a combination of states.

Will this work here?

What about:      Try path 1. If it accepts, accept. Else  
                          Try path 2. If it accepts, accept. Else

- 
-

### The Construction

At any point in the operation of a nondeterministic machine  $M$ , the maximum number of branches is

$$r = \frac{|K|}{\text{states}} \cdot (|\Sigma| + 2) \text{ actions}$$

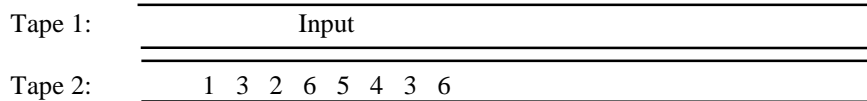
So imagine a table:

	1	2	3		r
(q1,σ1)	(p-,σ-)	(p-,σ-)	(p-,σ-)	(p-,σ-)	(p-,σ-)
(q1,σ2)	(p-,σ-)	(p-,σ-)	(p-,σ-)	(p-,σ-)	(p-,σ-)
(q1,σn)					
(q2,σ1)					
(q K ,σn)					

Note that if, in some configuration, there are not  $r$  different legal things to do, then some of the entries on that row will repeat.

### The Construction, Continued

$M_d$ : (suppose  $r = 6$ )



- $M_d$  chooses its 1st move from column 1
- $M_d$  chooses its 2nd move from column 3
- $M_d$  chooses its 3rd move from column 2

•  
•

until there are no more numbers on Tape 2

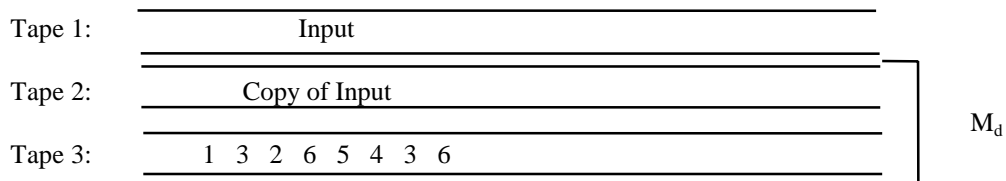
$M_d$  either:

- discovers that  $M$  would accept, or
- comes to the end of Tape 2.

In either case, it halts.

### The Construction, Continued

$M'$  (the machine that simulates  $M$ ):



Steps of  $M'$ :

- write  $\epsilon$  on Tape 3
- until  $M_d$  accepts do
  - (1) copy Input from Tape 1 to Tape 2
  - (2) run  $M_d$
  - (3) if  $M_d$  accepts, exit
  - (4) otherwise, generate lexicographically next string on Tape 3.

Pass	1	2	3		7	8	9		
Tape3	$\epsilon$	1	2	...	6	11	12	...	2635

## Nondeterministic Algorithms

### Other Turing Machine Extensions

Multiple heads (on one tape)

Emulation strategy: Use tracks to keep track of tape heads. (See book)

Multiple tapes, multiple heads

Emulation strategy: Use tracks to keep track of tapes and tape heads.

Two-dimensional semi-infinite “tape”

Emulation strategy: Use diagonal enumeration of two-dimensional grid. Use second tape to help you keep track of where the tape head is. (See book)

Two-dimensional infinite “tape” (really a sheet)

Emulation strategy: Use modified diagonal enumeration as with the semi-infinite case.

### What About Turing Machine Restrictions?

Can we make Turing machines even more limited and still get all the power?

Example:

We allow a tape alphabet of arbitrary size. What happens if we limit it to:

- One character?
- Two characters?
- Three characters?

# Problem Encoding, TM Encoding, and the Universal TM

Read K & S 5.1 & 5.2.

## Encoding a Problem as a Language

A Turing Machines deciding a language is analogous to the TM solving a **decision problem**.

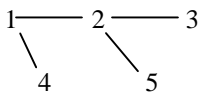
**Problem:** Is the number  $n$  prime?

**Instance of the problem:** Is the number 9 prime?

**Encoding of the problem,  $\langle n \rangle$ :**  $n$  as a binary number. Example: 1001

**Problem:** Is an undirected graph  $G$  connected?

**Instance of the problem:** Is the following graph connected?



**Encoding of the problem,  $\langle G \rangle$ :**

- 1)  $|V|$  as a binary number
- 2) A list of edges represented by pairs of binary numbers being the vertex numbers that the edge connects
- 3) All such binary numbers are separated by “/”.

Example: 101/1/10/10/11/1/100/10/101

## Problem View vs. Language View

**Problem View:** It is *unsolvable* whether a Turing Machine halts on a given input. This is called the **Halting Problem**.

**Language View:** Let  $H = \{ \langle M, w \rangle : \text{TM } M \text{ halts on input string } w \}$

$H$  is recursively enumerable but not recursive.

## The Universal Turing Machine

**Problem:** All our machines so far are hardwired.

**Question:** Does it make sense to talk about a programmable Turing machine that accepts as input

*program input string*

executes the program, and outputs

*output string*

Yes, it's called the Universal Turing Machine.

Notice that the Universal Turing machine semidecides  $H = \{ \langle M, w \rangle : \text{TM } M \text{ halts on input string } w \} = L(U)$ .

To define the Universal Turing Machine  $U$  we need to do two things:

1. Define an encoding operation for Turing machines.
2. Describe the operation of  $U$  given an input tape containing two inputs:
  - encoded Turing machine  $M$ ,
  - encoded input string to be given to  $M$ .

## Encoding a Turing Machine M

We need to describe  $M = (K, \Sigma, \delta, s, H)$  as a string. To do this we must:

1. Encode  $\delta$
2. Specify  $s$ .
3. Specify  $H$  (and  $y$  and  $n$ , if applicable)

1. To encode  $\delta$ , we need to:

1. Encode the states
2. Encode the tape alphabet
3. Specify the transitions

1.1 Encode the states as

$qs : s \in \{0, 1\}^+$  and

$|s| = i$  and

$i$  is the smallest integer such that  $2^i \geq |K|$

Example: 9 states  $i = 4$

$s = q0000$ ,

remaining states:  $q0001, q0010, q0011,$   
 $q0100, q0101, q0110, q0111, q1000$

### Encoding a Turing Machine M, Continued

1.2 Encode the tape alphabet as

$as : s \in \{0, 1\}^+$  and

$|s| = j$  and

$j$  is the smallest integer such that  $2^j \geq |\Sigma| + 2$  (the + 2 allows for  $\leftarrow$  and  $\rightarrow$ )

Example:  $\Sigma = \{\diamond, \square, a, b\}$   $j = 3$

$\square = a000$

$\diamond = a001$

$\leftarrow = a010$

$\rightarrow = a011$

$a = a100$

$b = a101$

### Encoding a Turing Machine M, Continued

1.3 Specify transitions as (state, input, state, output)

*Example:*  $(q00, a000, q11, a000)$

2. Specify  $s$  as  $q0^i$

3. Specify  $H$ :

- States with no transitions out are in  $H$ .
- If  $M$  decides a language, then  $H = \{y, n\}$ , and we will adopt the convention that  $y$  is the lexicographically smaller of the two states.

$y = q010$   $n = q011$

### Encoding Input Strings

We encode input strings to a machine  $M$  using the same character encoding we use for  $M$ .

For example, suppose that we are using the following encoding for symbols in  $M$ :

symbol	representation
$\square$	a000
$\diamond$	a001
$\leftarrow$	a010
$\rightarrow$	a011
a	a100

Then we would represent the string  $s = \diamond a \square a$  as  $\langle s \rangle = a001a100a100a000a100$

### An Encoding Example

Consider  $M = (\{s, q, h\}, \{\square, \diamond, a\}, \delta, s, \{h\})$ , where  $\delta =$

state	symbol	$\delta$
s	a	(q, $\square$ )
s	$\square$	(h, $\square$ )
s	$\diamond$	(s, $\rightarrow$ )
q	a	(s, a)
q	$\square$	(s, $\rightarrow$ )
q	$\diamond$	(q, $\rightarrow$ )

state/symbol	representation
s	q00
q	q01
h	q11
$\square$	a000
$\diamond$	a001
$\leftarrow$	a010
$\rightarrow$	a011
a	a100

The representation of  $M$ , denoted, " $M$ ",  $\langle M \rangle$ , or sometimes  $\rho(M) =$   
 $(q00, a100, q01, a000)$ ,  $(q00, a000, q11, a000)$ ,  $(q00, a001, q00, a011)$ ,  
 $(q01, a100, q00, a100)$ ,  $(q01, a000, q00, a011)$ ,  $(q01, a001, q01, a011)$

### Another Win of Encoding

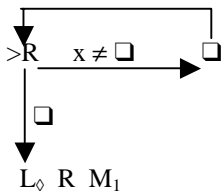
One big win of defining a way to encode any Turing machine  $M$ :

- It will make sense to talk about operations on programs (Turing machines). In other words, we can talk about some Turing machine  $T$  that takes another Turing machine (say  $M_1$ ) as input and transforms it into a different machine (say  $M_2$ ) that performs some different, but possibly related task.

#### Example of a transforming TM $T$ :

**Input:** a machine  $M_1$  that reads its input tape and performs some operation  $P$  on it.

**Output:** a machine  $M_2$  that performs  $P$  on an empty input tape:



### The Universal Turing Machine

The specification for  $U$ :

$$U("M" \ "w") = "M(w)"$$

$\diamond$	"M-----"			M"	"w-----"			$\square$	$\square$	
	1	0	0	0	0	0	0			
	$\square$	$\square$	$\square$	$\square$	$\square$	$\square$	$\square$			$\square$
	$\square$	$\square$	$\square$	$\square$	$\square$	$\square$	$\square$			$\square$
	$\square$	$\square$	$\square$	$\square$	$\square$	$\square$	$\square$			$\square$
$\diamond$	" $\diamond$ $\square$ "	"w-----"		w"	$\square$	$\square$		$\square$	$\square$	
	1	0	0	0	0	0	0			
	"M-----"			M"	$\square$	$\square$	$\square$			
	1	0	0	0	0	0	0			
	q	0	0	0	$\square$	$\square$	$\square$			
1	$\square$	$\square$	$\square$	$\square$	$\square$	$\square$	$\square$			

Initialization of  $U$ :

- Copy " $M$ " onto tape 2
- Insert " $\diamond \square$ " at the left edge of tape 1, then shift  $w$  over.
- Look at " $M$ ", figure out what  $i$  is, and write the encoding of state  $s$  on tape 3.

## The Operation of U

◇	a	0	0	1	a	0	0				
	1	0	0	0	0	0	0				
	"M-----M"				□	□	□			□	□
	1	0	0	0	0	0	0				
	q	0	0	0	□	□	□				
	1	□	□	□	□	□	□				

Simulate the steps of M:

1. Start with the heads:
  - tape 1: the a of the character being scanned,
  - tape 2: far left
  - tape 3: far left
2. Simulate one step:
  1. Scan tape 2 for a quadruple that matches current state, input pair.
  2. Perform the associated action, by changing tapes 1 and 3. If necessary, extend the tape.
  3. If no quadruple found, halt. Else go back to 2.

### An Example

Tape 1: a001a000a100a100a000a100

◇ □ a a □ a  
↑

Tape 2: (q00,a000,q11,a000), (q00,a001,q00,a011),  
 (q00,a100,q01,a000), (q01,a000,q00,a011),  
 (q01,a001,q01,a011), (q01,a100,q00,a100)

Tape 3: q01

↑

Result of simulating the next step:

Tape 1: a001a000a100a100a000a100

◇ □ a a □ a  
↑

Tape 3: q00

↑

### If A Universal Machine is Such a Good Idea ...

Could we define a Universal Finite State Machine?

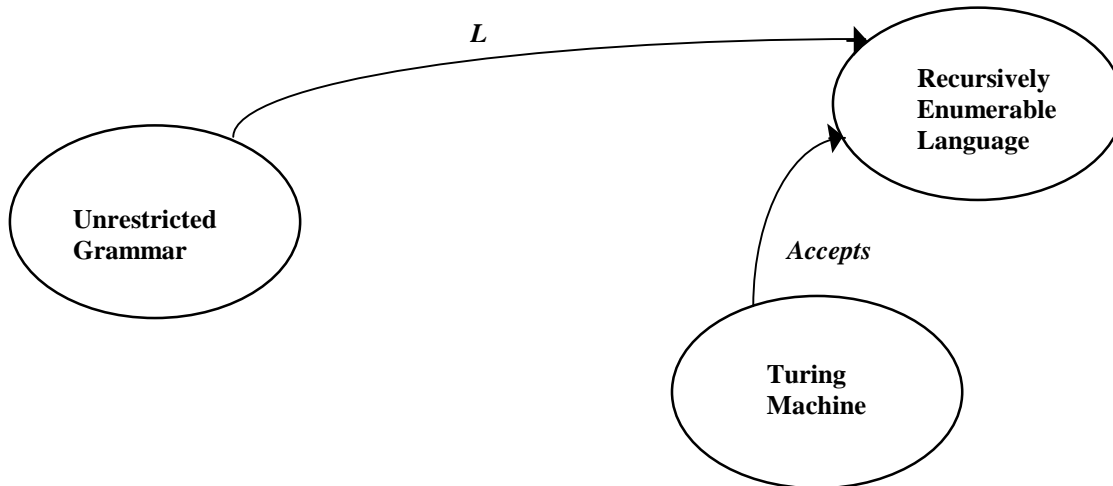
Such a FSM would accept the language

$$L = \{ "F" "w" : F \text{ is a finite state machine, and } w \in L(F) \}$$

# Grammars and Turing Machines

Do Homework 20.

## Grammars, Recursively Enumerable Languages, and Turing Machines



### Unrestricted Grammars

An unrestricted, or Type 0, or phrase structure grammar  $G$  is a quadruple  $(V, \Sigma, R, S)$ , where

- $V$  is an alphabet,
- $\Sigma$  (the set of terminals) is a subset of  $V$ ,
- $R$  (the set of rules) is a finite subset of
  - $(V^* \times V^*) \setminus (V^* \times \Sigma^*)$
- $S$  (the start symbol) is an element of  $V - \Sigma$ .

We define derivations just as we did for context-free grammars.

The language generated by  $G$  is

$$\{w \in \Sigma^* : S \Rightarrow_G^* w\}$$

There is no notion of a derivation tree or rightmost/leftmost derivation for unrestricted grammars.

### Unrestricted Grammars

Example:  $L = \{a^n b^n c^n, n > 0\}$

- $S \rightarrow aBSc$
- $S \rightarrow aBc$
- $Ba \rightarrow aB$
- $Bc \rightarrow bc$
- $Bb \rightarrow bb$

### Another Example

$L = \{w \in \{a, b, c\}^+ : \text{number of a's, b's and c's is the same}\}$

- $S \rightarrow ABCS$
- $S \rightarrow ABC$
- $AB \rightarrow BA$
- $BC \rightarrow CB$
- $AC \rightarrow CA$
- $BA \rightarrow AB$

- $CA \rightarrow AC$
- $CB \rightarrow BC$
- $A \rightarrow a$
- $B \rightarrow b$
- $C \rightarrow c$



## A Strong Procedural Feel

Unrestricted grammars have a procedural feel that is absent from restricted grammars.

Derivations often proceed in phases. We make sure that the phases work properly by using nonterminals as flags that we're in a particular phase.

It's very common to have two main phases:

- Generate the right number of the various symbols.
- Move them around to get them in the right order.

No surprise: unrestricted grammars are general computing devices.

### Equivalence of Unrestricted Grammars and Turing Machines

**Theorem:** A language is generated by an unrestricted grammar if and only if it is recursively enumerable (i.e., it is semidecided by some Turing machine  $M$ ).

**Proof:**

Only if (grammar  $\rightarrow$  TM): by construction of a nondeterministic Turing machine.

If (TM  $\rightarrow$  grammar): by construction of a grammar that mimics backward computations of  $M$ .

#### Proof that Grammar $\rightarrow$ Turing Machine

Given a grammar  $G$ , produce a Turing machine  $M$  that semidecides  $L(G)$ .

$M$  will be nondeterministic and will use two tapes:

$\diamond$	$\diamond$	$\square$	a	b	a	$\square$	$\square$	$\square$	$\square$	
	0	1	0	0	0	0	0			
	$\diamond$	a	S	T	a	b	$\square$			
	0	1	0	0	0	0	0			

For each nondeterministic "incarnation":

- Tape 1 holds the input.
- Tape 2 holds the current state of a proposed derivation.

At each step,  $M$  nondeterministically chooses a rule to try to apply and a position on tape 2 to start looking for the left hand side of the rule. Or it chooses to check whether tape 2 equals tape 1. If any such machine succeeds, we accept. Otherwise, we keep looking.

## Proof that Turing Machine $\rightarrow$ Grammar

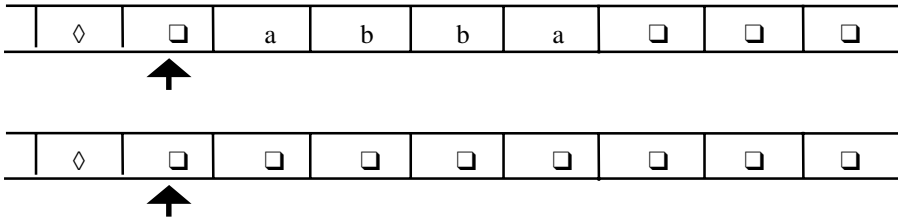
Suppose that  $M$  semidecides a language  $L$  (it halts when fed strings in  $L$  and loops otherwise). Then we can build  $M'$  that halts in the configuration  $(h, \diamond \square)$ .

We will define  $G$  so that it simulates  $M'$  backwards.

We will represent the configuration  $(q, \diamond u \underline{a} w)$  as

$\triangleright u a q w \triangleleft$

$M'$   
goes from



Then, if  $w \in L$ , we require that our grammar produce a derivation of the form

$S \Rightarrow_G \triangleright \square h \triangleleft$  (produces final state of  $M'$ )  
 $\Rightarrow_{G^*} \triangleright \square a b q \triangleleft$  (some intermediate state of  $M'$ )  
 $\Rightarrow_{G^*} \triangleright \square s w \triangleleft$  (the initial state of  $M'$ )  
 $\Rightarrow_G w \triangleleft$  (via a special rule to clean up  $\triangleright \square s$ )  
 $\Rightarrow_G w$  (via a special rule to clean up  $\triangleleft$ )

### The Rules of $G$

$S \rightarrow \triangleright \square h \triangleleft$  (the halting configuration)

$\triangleright \square s \rightarrow \epsilon$  (clean-up rules to be applied at the end)

$\triangleleft \rightarrow \epsilon$

Rules that correspond to  $\delta$ :

If  $\delta(q, a) = (p, b)$  :  $bp \rightarrow aq$

If  $\delta(q, a) = (p, \rightarrow)$  :  $abp \rightarrow aqb \quad \forall b \in \Sigma$   
 $a \square p \triangleleft \rightarrow aq \triangleleft$

If  $\delta(q, a) = (p, \leftarrow)$ ,  $a \neq \square$  :  $pa \rightarrow aq$

If  $\delta(q, \square) = (p, \leftarrow)$  :  $p \square b \rightarrow \square qb \quad \forall b \in \Sigma$   
 $p \triangleleft \rightarrow \square q \triangleleft$

### A REALLY Simple Example

$M' = (K, \{a\}, \delta, s, \{h\})$ , where

$\delta = \{$	$((s, \square), (q, \rightarrow)),$	1
	$((q, a), (q, \rightarrow)),$	2
	$((q, \square), (t, \leftarrow)),$	3
	$((t, a), (p, \square)),$	4
	$((t, \square), (h, \square)),$	5
	$((p, \square), (t, \leftarrow))$	6

$L = a^*$

<p><math>S \rightarrow \square h &lt;</math></p> <p><math>\square s \rightarrow \epsilon</math></p> <p><math>&lt; \rightarrow \epsilon</math></p> <p>(1) <math>\square \square q \rightarrow \square s \square</math></p> <p><math>\square a q \rightarrow \square s a</math></p> <p><math>\square \square q &lt; \rightarrow \square s &lt;</math></p> <p>(2) <math>a \square q \rightarrow a q \square</math></p> <p><math>aaq \rightarrow aqa</math></p> <p><math>a \square q &lt; \rightarrow aq &lt;</math></p>	<p>(3)</p> <p>(4)</p> <p>(5)</p> <p>(6)</p>	<p><math>t \square \square \rightarrow \square q \square</math></p> <p><math>t \square a \rightarrow \square qa</math></p> <p><math>t &lt; \rightarrow \square q &lt;</math></p> <p><math>\square p \rightarrow at</math></p> <p><math>\square h \rightarrow \square t</math></p> <p><math>t \square \square \rightarrow \square p \square</math></p> <p><math>t \square a \rightarrow \square pa</math></p> <p><math>t &lt; \rightarrow \square p &lt;</math></p>
--	---	--

#### Working It Out

<p><math>S \rightarrow \square h &lt;</math></p> <p><math>\square s \rightarrow \epsilon</math></p> <p><math>&lt; \rightarrow \epsilon</math></p> <p>(1) <math>\square \square q \rightarrow \square s \square</math></p> <p><math>\square a q \rightarrow \square s a</math></p> <p><math>\square \square q &lt; \rightarrow \square s &lt;</math></p> <p>(2) <math>a \square q \rightarrow a q \square</math></p> <p><math>aaq \rightarrow aqa</math></p> <p><math>a \square q &lt; \rightarrow aq &lt;</math></p>	<p>1</p> <p>2</p> <p>3</p> <p>4</p> <p>5</p> <p>6</p> <p>7</p> <p>8</p> <p>9</p>	<p>(3) <math>t \square \square \rightarrow \square q \square</math></p> <p><math>t \square a \rightarrow \square qa</math></p> <p><math>t &lt; \rightarrow \square q &lt;</math></p> <p>(4) <math>\square p \rightarrow at</math></p> <p>(5) <math>\square h \rightarrow \square t</math></p> <p>(6) <math>t \square \square \rightarrow \square p \square</math></p> <p><math>t \square a \rightarrow \square pa</math></p> <p><math>t &lt; \rightarrow \square p &lt;</math></p>	<p>10</p> <p>11</p> <p>12</p> <p>13</p> <p>14</p> <p>15</p> <p>16</p> <p>17</p>
--	--	--	---

---

<p><math>\square s a a &lt;</math></p> <p><math>\square a q a &lt;</math></p> <p><math>\square a a q &lt;</math></p> <p><math>\square a a \square q &lt;</math></p> <p><math>\square a a t &lt;</math></p> <p><math>\square a \square p &lt;</math></p> <p><math>\square a t &lt;</math></p> <p><math>\square \square p &lt;</math></p> <p><math>\square t &lt;</math></p> <p><math>\square h &lt;</math></p>	<p>1</p> <p>2</p> <p>2</p> <p>3</p> <p>4</p> <p>6</p> <p>4</p> <p>6</p> <p>5</p> <p>5</p>	<p>S</p>	<p><math>\Rightarrow \square h &lt;</math></p> <p><math>\Rightarrow \square t &lt;</math></p> <p><math>\Rightarrow \square \square p &lt;</math></p> <p><math>\Rightarrow \square a t &lt;</math></p> <p><math>\Rightarrow \square a \square p &lt;</math></p> <p><math>\Rightarrow \square a a t &lt;</math></p> <p><math>\Rightarrow \square a a \square q &lt;</math></p> <p><math>\Rightarrow \square a a q &lt;</math></p> <p><math>\Rightarrow \square a q a &lt;</math></p> <p><math>\Rightarrow \square s a a &lt;</math></p> <p><math>\Rightarrow a a &lt;</math></p> <p><math>\Rightarrow a a</math></p>	<p>1</p> <p>14</p> <p>17</p> <p>13</p> <p>17</p> <p>13</p> <p>12</p> <p>9</p> <p>8</p> <p>5</p> <p>2</p> <p>3</p>
---	---	----------	--	---

## An Alternative Proof

An alternative is to build a grammar  $G$  that simulates the forward operation of a Turing machine  $M$ . It uses alternating symbols to represent two interleaved tapes. One tape remembers the starting string, the other “working” tape simulates the run of the machine.

The first (generate) part of  $G$ :

Creates all strings over  $\Sigma^*$  of the form

$$w = \diamond \diamond \square \square Q_S a_1 a_1 a_2 a_2 a_3 a_3 \square \square \dots$$

The second (test) part of  $G$  simulates the execution of  $M$  on a particular string  $w$ . An example of a partially derived string:

$$\diamond \diamond \square \square a \ 1 \ b \ 2 \ c \ c \ b \ 4 \ Q_3 \ a \ 3$$

Examples of rules:

$$b \ b \ Q \ 4 \rightarrow b \ 4 \ Q \ 4 \quad (\text{rewrite } b \text{ as } 4)$$

$$b \ 4 \ Q \ 3 \rightarrow Q \ 3 \ b \ 4 \quad (\text{move left})$$

The third (cleanup) part of  $G$  erases the junk if  $M$  ever reaches  $h$ .

Example rule:

$$\# \ h \ a \ 1 \rightarrow a \ \# \ h \quad (\text{sweep } \# \ h \text{ to the right erasing the working “tape”})$$

## Computing with Grammars

We say that  $G$  **computes**  $f$  if, for all  $w, v \in \Sigma^*$ ,

$$SwS \Rightarrow_G^* v \quad \text{iff } v = f(w)$$

Example:

$$S1S \Rightarrow_G^* 11$$

$$S11S \Rightarrow_G^* 111 \quad f(x) = \text{succ}(x)$$

A function  $f$  is called **grammatically computable** iff there is a grammar  $G$  that computes it.

**Theorem:** A function  $f$  is recursive iff it is grammatically computable.

In other words, if a Turing machine can do it, so can a grammar.

### Example of Computing with a Grammar

$f(x) = 2x$ , where  $x$  is an integer represented in unary

$G = (\{S, 1\}, \{1\}, R, S)$ , where  $R =$

$$S1 \rightarrow 11S$$

$$SS \rightarrow \epsilon$$

Example:

Input: S111S

Output:

## More on Functions: Why Have We Been Using Recursive as a Synonym for Computable? Primitive Recursive Functions

Define a set of basic functions:

- $\text{zero}_k(n_1, n_2, \dots, n_k) = 0$
- $\text{identity}_{k,j}(n_1, n_2, \dots, n_k) = n_j$
- $\text{successor}(n) = n + 1$

Combining functions:

- Composition of  $g$  with  $h_1, h_2, \dots, h_k$  is  
 $g(h_1(\ ), h_2(\ ), \dots, h_k(\ ))$
- Primitive recursion of  $f$  in terms of  $g$  and  $h$ :  
 $f(n_1, n_2, \dots, n_k, 0) = g(n_1, n_2, \dots, n_k)$   
 $f(n_1, n_2, \dots, n_k, m+1) = h(n_1, n_2, \dots, n_k, m, f(n_1, n_2, \dots, n_k, m))$

Example:       $\text{plus}(n, 0) = n$   
 $\text{plus}(n, m+1) = \text{succ}(\text{plus}(n, m))$

### Primitive Recursive Functions and Computability

Trivially true: all primitive recursive functions are Turing computable.  
 What about the other way: Not all Turing computable functions are primitive recursive.

**Proof:**

Lexicographically enumerate the unary primitive recursive functions,  $f_0, f_1, f_2, f_3, \dots$

Define  $g(n) = f_n(n) + 1$ .

$G$  is clearly computable, but it is not on the list. Suppose it were  $f_m$  for some  $m$ . Then

$$f_m(m) = f_m(m) + 1, \text{ which is absurd.}$$

	0	1	2	3	4
$f_0$					
$f_1$					
$f_2$					
$f_3$				27	
$f_4$					

Suppose  $g$  is  $f_3$ . Then  $g(3) = 27 + 1 = 28$ . Contradiction.

### Functions that Aren't Primitive Recursive

**Example:**      Ackermann's function:       $A(0, y) = y + 1$   
 $A(x + 1, 0) = A(x, 1)$   
 $A(x + 1, y + 1) = A(x, A(x + 1, y))$

	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>
<b>0</b>	1	2	3	4	5
<b>1</b>	2	3	4	5	6
<b>2</b>	3	5	7	9	11
<b>3</b>	5	13	29	61	125
<b>4</b>	13	65533	$2^{65536} - 3$ *	$2^{2^{65536}} - 3$ #	$2^{2^{2^{65536}}} - 3$ %

* 19,729 digits	10 <sup>17</sup> seconds since big bang
# 10 <sup>5940</sup> digits	10 <sup>87</sup> protons and neutrons
% 10 <sup>10<sup>5939</sup></sup> digits	10 <sup>-23</sup> light seconds = width of proton or neutron

Thus writing digits at the speed of light on all protons and neutrons in the universe (all lined up) starting at the big bang would have produced 10<sup>127</sup> digits.

## Recursive Functions

A function is  **$\mu$ -recursive** if it can be obtained from the basic functions using the operations of:

- Composition,
- Recursive definition, and
- Minimalization of minimalizable functions:

The **minimalization** of  $g$  (of  $k + 1$  arguments) is a function  $f$  of  $k$  arguments defined as:

$$f(n_1, n_2, \dots, n_k) = \begin{cases} \text{the least } m \text{ such that } g(n_1, n_2, \dots, n_k, m) = 1, & \text{if such an } m \text{ exists,} \\ 0 & \text{otherwise} \end{cases}$$

A function  $g$  is **minimalizable** iff for every  $n_1, n_2, \dots, n_k$ , there is an  $m$  such that  $g(n_1, n_2, \dots, n_k, m) = 1$ .

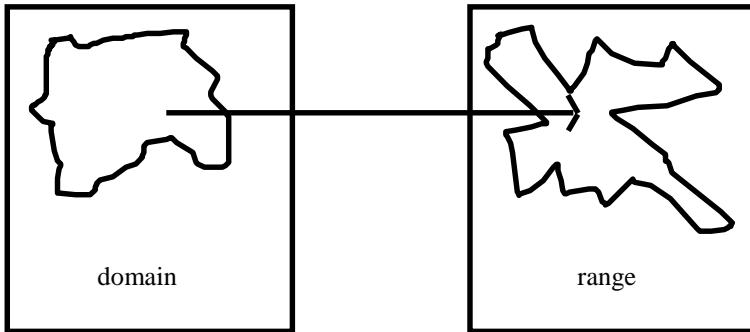
**Theorem:** A function is  $\mu$ -recursive iff it is recursive (i.e., computable by a Turing machine).

## Partial Recursive Functions

Consider the following function  $f$ :

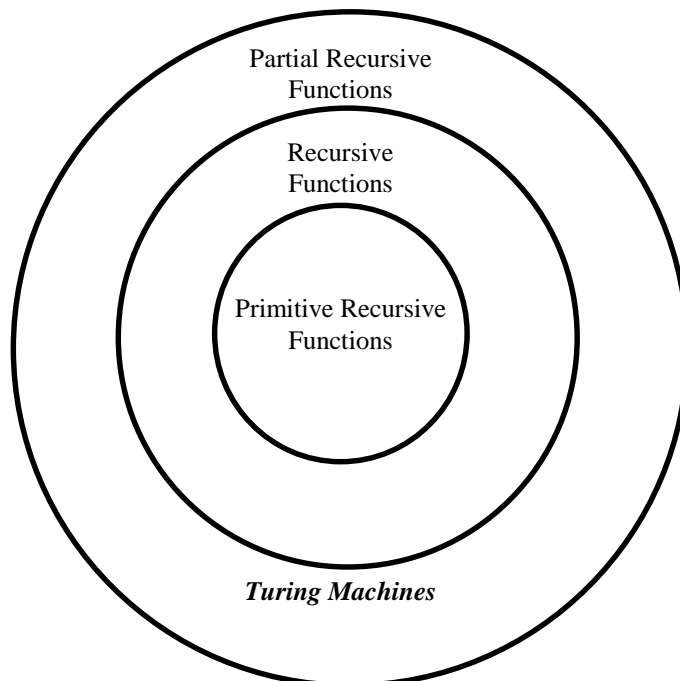
$$f(n) = \begin{cases} 1 & \text{if TM}(n) \text{ halts on a blank tape} \\ 0 & \text{otherwise} \end{cases}$$

The domain of  $f$  is the natural numbers. Is  $f$  recursive?

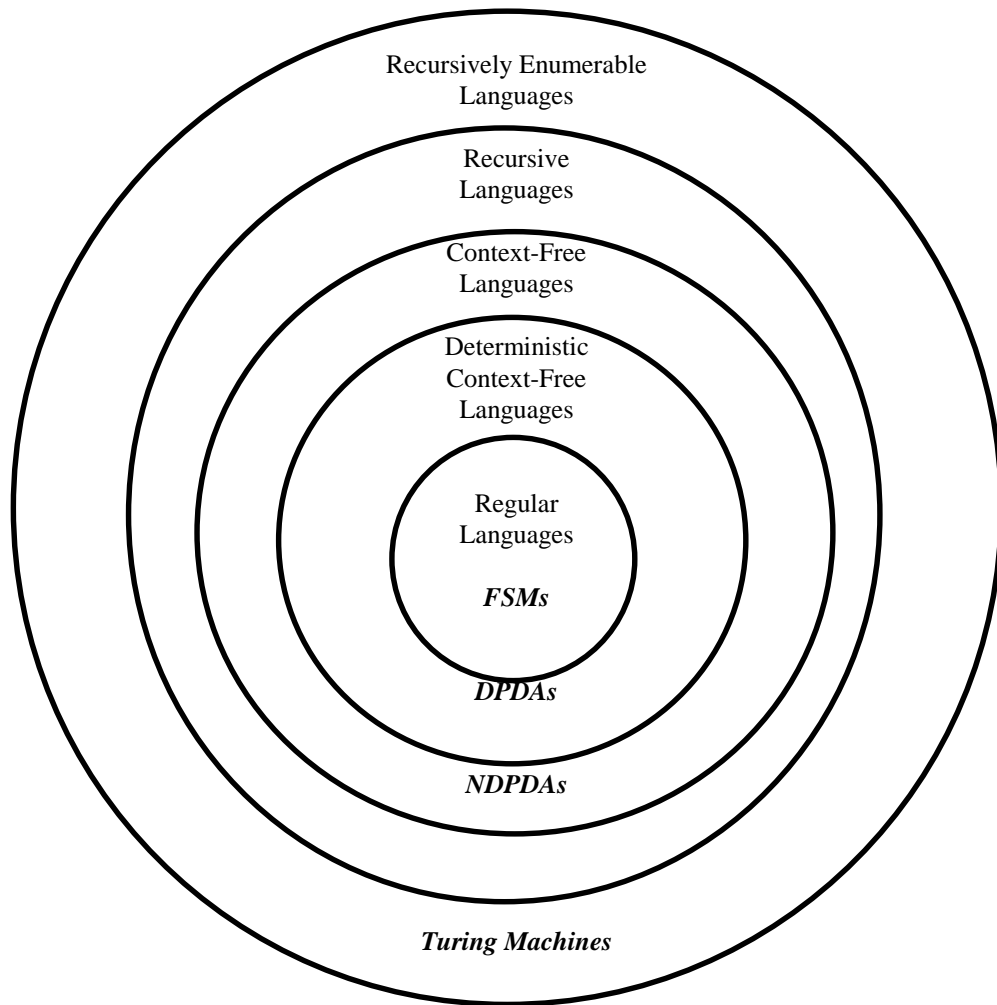


**Theorem:** There are uncountably many partially recursive functions (but only countably many Turing machines).

## Functions and Machines



## Languages and Machines



### Is There Anything In Between CFGs and Unrestricted Grammars?

Answer: yes, various things have been proposed.

#### Context-Sensitive Grammars and Languages:

A grammar  $G$  is context sensitive if all productions are of the form

$$x \rightarrow y$$

and  $|x| \leq |y|$

In other words, there are no length-reducing rules.

A language is context sensitive if there exists a context-sensitive grammar for it.

Examples:

$$L = \{a^n b^n c^n, n > 0\}$$

$$L = \{w \in \{a, b, c\}^+ : \text{number of a's, b's and c's is the same}\}$$

## Context-Sensitive Languages are Recursive

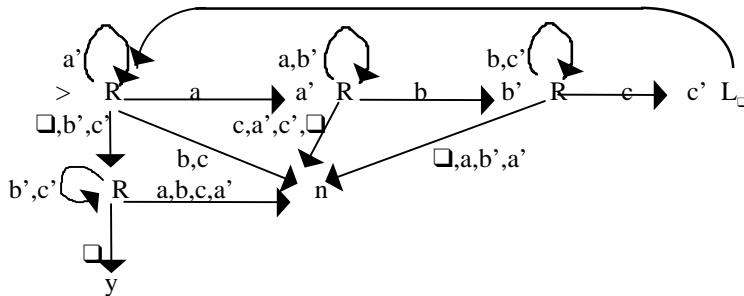
The basic idea: To decide if a string  $w$  is in  $L$ , start generating strings systematically, shortest first. If you generate  $w$ , accept. If you get to strings that are longer than  $w$ , reject.

### Linear Bounded Automata

A linear bounded automaton is a nondeterministic Turing machine the length of whose tape is bounded by some fixed constant  $k$  times the length of the input.

Example:  $L = \{a^n b^n c^n : n \geq 0\}$

$\diamond \square aabbcc \square \square \square \square \square \square \square \square$



### Context-Sensitive Languages and Linear Bounded Automata

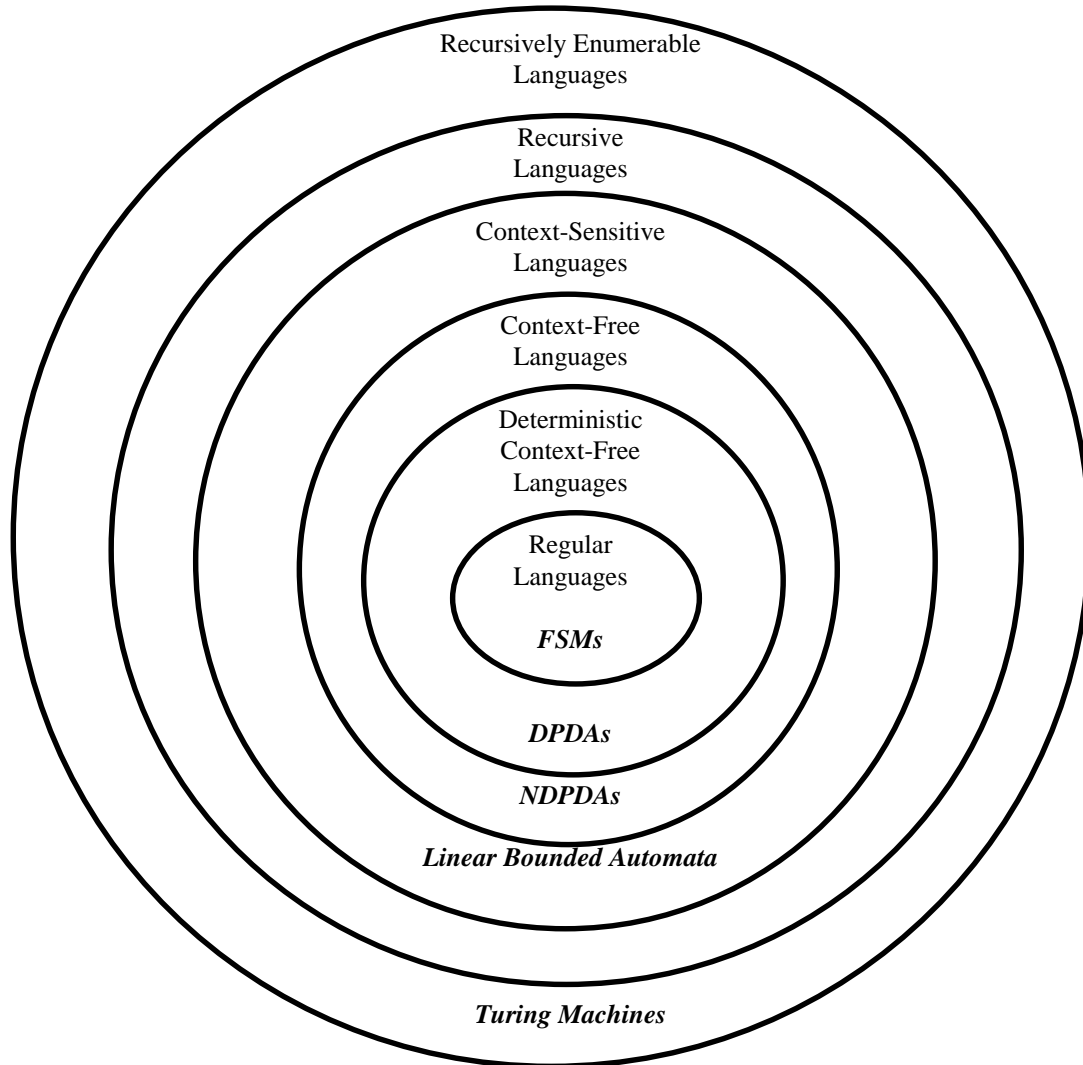
**Theorem:** The set of context-sensitive languages is exactly the set of languages that can be accepted by linear bounded automata.

**Proof:** (sketch) We can construct a linear-bounded automaton  $B$  for any context-sensitive language  $L$  defined by some grammar  $G$ . We build a machine  $B$  with a two track tape. On input  $w$ ,  $B$  keeps  $w$  on the first tape. On the second tape, it nondeterministically constructs all derivations of  $G$ . The key is that as soon as any derivation becomes longer than  $|w|$  we stop, since we know it can never get any shorter and thus match  $w$ . There is also a proof that from any lba we can construct a context-sensitive grammar, analogous to the one we used for Turing machines and unrestricted grammars.

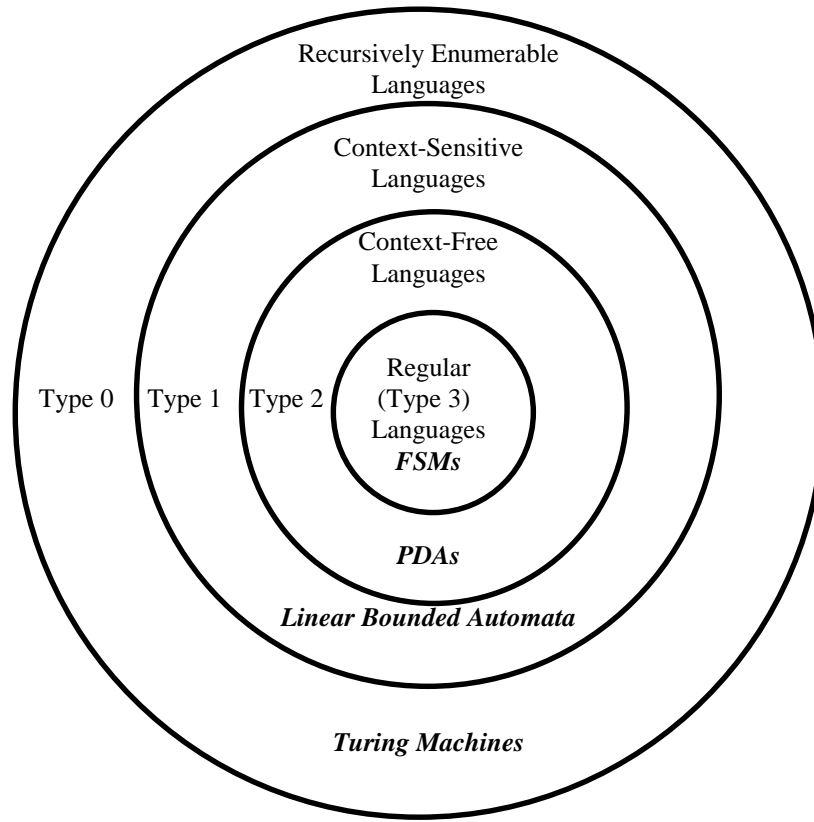
**Theorem:** There exist recursive languages that are not context sensitive.



## Languages and Machines



## The Chomsky Hierarchy



# Undecidability

Read K & S 5.1, 5.3, & 5.4.

Read Supplementary Materials: Recursively Enumerable Languages, Turing Machines, and Decidability.

Do Homeworks 21 & 22.

## Church's Thesis (Church-Turing Thesis)

An **algorithm** is a formal procedure that halts.

The Thesis: Anything that can be computed by any algorithm can be computed by a Turing machine.

Another way to state it: All "reasonable" formal models of computation are equivalent to the Turing machine.

This isn't a formal statement, so we can't prove it. But many different computational models have been proposed and they all turn out to be equivalent.

Examples:

- unrestricted grammars
- lambda calculus
- cellular automata
- DNA computing
- quantum computing (?)

## The Unsolvability of the Halting Problem

Suppose we could implement the decision procedure

```
HALTS(M, x)
  M: string representing a Turing Machine
  x: string representing the input for M
  If M(x) halts then True
  else False
```

Then we could define

```
TROUBLE(x)
  x: string
  If HALTS(x, x) then loop forever
  else halt
```

So now what happens if we invoke TROUBLE("TROUBLE"), which invokes HALTS("TROUBLE", "TROUBLE")

If HALTS says that TROUBLE halts on itself then TROUBLE loops. If HALTS says that TROUBLE loops, then TROUBLE halts. Either way, we reach a contradiction, so HALTS(M, x) cannot be made into a decision procedure.

## Another View

**The Problem View:** The halting problem is undecidable.

**The Language View:** Let  $H = \{ \langle M \rangle \langle w \rangle : \text{TM } M \text{ halts on input string } w \}$   
 $H$  is recursively enumerable but not recursive.

Why?

$H$  is recursively enumerable because it can be semidecided by  $U$ , the Universal Turing Machine.

But  $H$  cannot be recursive. If it were, then it would be decided by some TM  $M_H$ . But  $M_H(\langle M \rangle \langle w \rangle)$  would have to be:  
If  $M$  is not a syntactically valid TM, then False.  
else HALTS( $\langle M \rangle \langle w \rangle$ )

But we know cannot that HALTS cannot exist.

## If H were Recursive

$H = \{ \langle M \rangle \langle w \rangle : \text{TM } M \text{ halts on input string } w \}$

**Theorem:** If  $H$  were also recursive, then every recursively enumerable language would be recursive.

**Proof:** Let  $L$  be any RE language. Since  $L$  is RE, there exists a TM  $M$  that semidecides it.

Suppose  $H$  is recursive and thus is decided by some TM  $O$  (oracle).

We can build a TM  $M'$  from  $M$  that decides  $L$ :

1.  $M'$  transforms its input tape from  $\langle w \rangle$  to  $\langle M \rangle \langle w \rangle$ .
2.  $M'$  invokes  $O$  on its tape and returns whatever answer  $O$  returns.

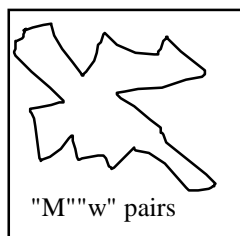
So, if  $H$  were recursive, all RE languages would be. **But it isn't.**

## Undecidable Problems, Languages that Are Not Recursive, and Partial Functions

**The Problem View:** The halting problem is undecidable.

**The Language View:** Let  $H = \{ \langle M \rangle \langle w \rangle : \text{TM } M \text{ halts on input string } w \}$   
 $H$  is recursively enumerable but not recursive.

**The Functional View:** Let  $f(w) = M(w)$   
 $f$  is a partial function on  $\Sigma^*$



## Other Undecidable Problems About Turing Machines

- Given a Turing machine  $M$ , does  $M$  halt on the empty tape?
- Given a Turing machine  $M$ , is there any string on which  $M$  halts?
- Given a Turing machine  $M$ , does  $M$  halt on every input string?
- Given two Turing machines  $M_1$  and  $M_2$ , do they halt on the same input strings?
- Given a Turing machine  $M$ , is the language that  $M$  semidecides regular? Is it context-free? Is it recursive?

### Post Correspondence Problem

Consider two lists of strings over some alphabet  $\Sigma$ . The lists must be finite and of equal length.

$$A = x_1, x_2, x_3, \dots, x_n$$

$$B = y_1, y_2, y_3, \dots, y_n$$

Question: Does there exist some finite sequence of integers that can be viewed as indexes of  $A$  and  $B$  such that, when elements of  $A$  are selected as specified and concatenated together, we get the same string we get when elements of  $B$  are selected also as specified?

For example, if we assert that 1, 3, 4 is such a sequence, we're asserting that  $x_1x_3x_4 = y_1y_3y_4$

Any problem of this form is an instance of the Post Correspondence Problem.

Is the Post Correspondence Problem decidable?

### Post Correspondence Problem Examples

i	A	B
1	1	111
2	10111	10
3	10	0

i	A	B
1	10	101
2	011	11
3	101	011

### Some Languages Aren't Even Recursively Enumerable

A pragmatically non RE language:  $L_1 = \{ (i, j) : i, j \text{ are integers where the low order five digits of } i \text{ are a street address number and } j \text{ is the number of houses with that number on which it rained on November 13, 1946} \}$

An analytically non RE language:  $L_2 = \{ x : x = "M" \text{ of a Turing machine } M \text{ and } M("M") \text{ does not halt} \}$

Why isn't  $L_2$  RE? Suppose it were. Then there would be a TM  $M^*$  that semidecides  $L_2$ . Is " $M^*$ " in  $L_2$ ?

- If it is, then  $M^*(\text{"M*"})$  halts (by the definition of  $M^*$  as a semideciding machine for  $L_2$ )
- But, by the definition of  $L_2$ , if " $M^*$ "  $\in L_2$ , then  $M^*(\text{"M*"})$  does not halt.

Contradiction. So  $L_2$  is not RE.

### Another Non RE Language

$\overline{H}$

Why not?

## Reduction

Let  $L_1, L_2 \subseteq \Sigma^*$  be languages. A **reduction** from  $L_1$  to  $L_2$  is a recursive function  $\tau: \Sigma^* \rightarrow \Sigma^*$  such that  $x \in L_1$  iff  $\tau(x) \in L_2$ .

Example:

$$L_1 = \{a, b : a, b \in \mathbb{N} : b = a + 1\}$$

$$\Downarrow \quad \tau = \text{Succ}$$

$$\Downarrow \quad a, b \text{ becomes } \text{Succ}(a), b$$

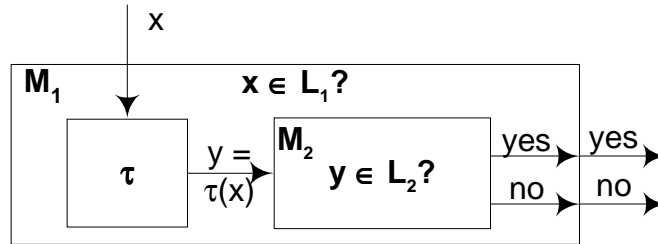
$$L_2 = \{a, b : a, b \in \mathbb{N} : a = b\}$$

If there is a Turing machine  $M_2$  to decide  $L_2$ , then I can build a Turing machine  $M_1$  to decide  $L_1$ :

1. Take the input and apply Succ to the first number.
2. Invoke  $M_2$  on the result.
3. Return whatever answer  $M_2$  returns.

### Reductions and Recursive Languages

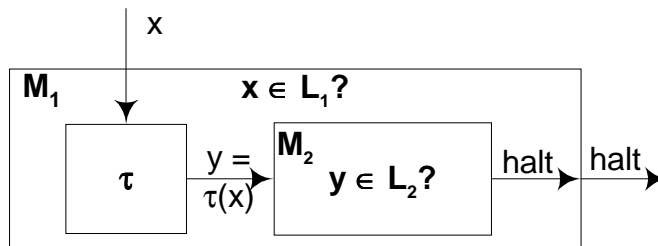
**Theorem:** If there is a reduction from  $L_1$  to  $L_2$  and  $L_2$  is recursive, then  $L_1$  is recursive.



**Theorem:** If there is a reduction from  $L_1$  to  $L_2$  and  $L_1$  is not recursive, then  $L_2$  is not recursive.

### Reductions and RE Languages

**Theorem:** If there is a reduction from  $L_1$  to  $L_2$  and  $L_2$  is RE, then  $L_1$  is RE.



**Theorem:** If there is a reduction from  $L_1$  to  $L_2$  and  $L_1$  is not RE, then  $L_2$  is not RE.

### Can it be Decided if M Halts on the Empty Tape?

This is equivalent to, "Is the language  $L_2 = \{ \langle M \rangle : \text{Turing machine } M \text{ halts on the empty tape} \}$  recursive?"

$$L_1 = H = \{ s = \langle M \rangle \langle w \rangle : \text{Turing machine } M \text{ halts on input string } w \}$$

$$\Downarrow \qquad \tau$$

(?M<sub>2</sub>)  $L_2 = \{ s = \langle M \rangle : \text{Turing machine } M \text{ halts on the empty tape} \}$

Let  $\tau$  be the function that, from  $\langle M \rangle$  and  $\langle w \rangle$ , constructs  $\langle M^* \rangle$ , which operates as follows on an empty input tape:

1. Write  $w$  on the tape.
2. Operate as  $M$  would have.

If  $M_2$  exists, then  $M_1 = M_2(M_\tau(s))$  decides  $L_1$ .

### A Formal Reduction Proof

Prove that  $L_2 = \{ \langle M \rangle : \text{Turing machine } M \text{ halts on the empty tape} \}$  is not recursive.

Proof that  $L_2$  is not recursive via a reduction from  $H = \{ \langle M, w \rangle : \text{Turing machine } M \text{ halts on input string } w \}$ , a non-recursive language. Suppose that there exists a TM,  $M_2$  that decides  $L_2$ . Construct a machine to decide  $H$  as  $M_1(\langle M, w \rangle) = M_2(\tau(\langle M, w \rangle))$ . The  $\tau$  function creates from  $\langle M \rangle$  and  $\langle w \rangle$  a new machine  $M^*$ .  $M^*$  ignores its input and runs  $M$  on  $w$ , halting exactly when  $M$  halts on  $w$ .

- $\langle M, w \rangle \in H \Rightarrow M \text{ halts on } w \Rightarrow M^* \text{ always halts} \Rightarrow \epsilon \in L(M^*) \Rightarrow \langle M^* \rangle \in L_2 \Rightarrow M_2 \text{ accepts} \Rightarrow M_1 \text{ accepts.}$
- $\langle M, w \rangle \notin H \Rightarrow M \text{ does not halt on } w \Rightarrow \epsilon \notin L(M^*) \Rightarrow \langle M^* \rangle \notin L_2 \Rightarrow M_2 \text{ rejects} \Rightarrow M_1 \text{ rejects.}$

Thus, if there is a machine  $M_2$  that decides  $L_2$ , we could use it to build a machine that decides  $H$ . Contradiction.  $\therefore L_2$  is not recursive.

### Important Elements in a Reduction Proof

- A clear declaration of the reduction “from” and “to” languages and what you’re trying to prove with the reduction.
- A description of how a machine is being constructed for the “from” language based on an assumed machine for the “to” language and a recursive  $\tau$  function.
- A description of the  $\tau$  function’s inputs and outputs. If  $\tau$  is doing anything nontrivial, it is a good idea to argue that it is recursive.
- Note that machine diagrams are not necessary or even sufficient in these proofs. Use them as thought devices, where needed.
- Run through the logic that demonstrates how the “from” language is being decided by your reduction. You must do both accepting and rejecting cases.
- Declare that the reduction proves that your “to” language is not recursive.

### The Most Common Mistake: Doing the Reduction Backwards

The right way to use reduction to show that  $L_2$  is not recursive:

1. Given that  $L_1$  is not recursive,
2. Reduce  $L_1$  to  $L_2$ , i.e. show how to solve  $L_1$  (the known one) in terms of  $L_2$  (the unknown one)



Example: If there exists a machine  $M_2$  that solves  $L_2$ , the problem of deciding whether a Turing machine halts on a blank tape, then we could do  $H$  (deciding whether  $M$  halts on  $w$ ) as follows:

1. Create  $M^*$  from  $M$  such that  $M^*$ , given a blank tape, first writes  $w$  on its tape, then simulates the behavior of  $M$ .
2. Return  $M_2(\langle M^* \rangle)$ .

Doing it wrong by reducing  $L_2$  (the unknown one to  $L_1$ ): If there exists a machine  $M_1$  that solves  $H$ , then we could build a machine that solves  $L_2$  as follows:

1. Return  $(M_1(\langle M \rangle, \langle \epsilon \rangle))$ .

## Why Backwards Doesn't Work

Suppose that we have proved that the following problem  $L_1$  is unsolvable: Determine the number of days that have elapsed since the beginning of the universe.

Now consider the following problem  $L_2$ : Determine the number of days that had elapsed between the beginning of the universe and the assassination of Abraham Lincoln.

Reduce  $L_1$  to  $L_2$ :  
 $L_1 = L_2 + (\text{now} - 4/9/1865)$   $L_1$   
 $\Downarrow$   
 $L_2$

Reduce  $L_2$  to  $L_1$ :  
 $L_2 = L_1 - (\text{now} - 4/9/1865)$   $L_2$   
 $\Downarrow$   
 $L_1$

## Why Backwards Doesn't Work, Continued

$L_1$  = days since beginning of universe

$L_2$  = elapsed days between the beginning of the universe and the assassination of Abraham Lincoln.

$L_3$  = days between the assassination of Abraham Lincoln and now.

**Considering  $L_2$ :**  
 Reduce  $L_1$  to  $L_2$ :  
 $L_1 = L_2 + (\text{now} - 4/9/1865)$   $L_1$   
 $\Downarrow$   
 $L_2$

Reduce  $L_2$  to  $L_1$ :  
 $L_2 = L_1 - (\text{now} - 4/9/1865)$   $L_2$   
 $\Downarrow$   
 $L_1$

**Considering  $L_3$ :**  
 Reduce  $L_1$  to  $L_3$ :  
 $L_1 = \text{oops}$   $L_1$   
 $\Downarrow$   
 $L_3$

Reduce  $L_3$  to  $L_1$ :  
 $L_3 = L_1 - 365 - (\text{now} - 4/9/1866)$   $L_3$   
 $\Downarrow$   
 $L_1$

## Is There Any String on Which M Halts?

$$L_1 = H = \{s = "M" "w" : \text{Turing machine } M \text{ halts on input string } w\}$$

$$\Downarrow \quad \tau$$

$$(?M_2) \quad L_2 = \{s = "M" : \text{there exists a string on which Turing machine } M \text{ halts}\}$$

Let  $\tau$  be the function that, from "M" and "w", constructs "M\*", which operates as follows:

1. M\* examines its input tape.
2. If it is equal to w, then it simulates M.
3. If not, it loops.

Clearly the only input on which M\* has a chance of halting is w, which it does iff M would halt on w.

If  $M_2$  exists, then  $M_1 = M_2(M_\tau(s))$  decides  $L_1$ .



## Does M Halt on All Inputs?

$$L_1 = \{s = "M" : \text{Turing machine } M \text{ halts on the empty tape}\}$$

$$\Downarrow \quad \tau$$

$$(?M_2) \quad L_2 = \{s = "M" : \text{Turing machine } M \text{ halts on all inputs}\}$$

Let  $\tau$  be the function that, from "M", constructs "M\*", which operates as follows:

1. Erase the input tape.
2. Simulate M.

Clearly  $M^*$  either halts on all inputs or on none, since it ignores its input.

If  $M_2$  exists, then  $M_1 = M_2(M_\tau(s))$  decides  $L_1$ .

## Rice's Theorem

**Theorem:** No nontrivial property of the recursively enumerable languages is decidable.

**Alternate statement:** Let  $P: 2^{\Sigma^*} \rightarrow \{\text{true}, \text{false}\}$  be a nontrivial property of the recursively enumerable languages. The language  $\{ "M" : P(L(M)) = \text{True} \}$  is not recursive.

By "nontrivial" we mean a property that is not simply true for all languages or false for all languages.

### Examples:

- L contains only even length strings.
- L contains an odd number of strings.
- L contains all strings that start with "a".
- L is infinite.
- L is regular.

### Note:

Rice's theorem applies to languages, not machines. So, for example, the following properties of machines are decidable:

- M contains an even number of states
- M has an odd number of symbols in its tape alphabet

Of course, we need a way to define a language. We'll use machines to do that, but the properties we'll deal with are properties of  $L(M)$ , not of M itself.

## Proof of Rice's Theorem

**Proof:** Let P be any nontrivial property of the RE languages.

$$L_1 = H = \{s = "M" "w" : \text{Turing machine } M \text{ halts on input string } w\}$$

$$\Downarrow \quad \tau$$

$$(?M_2) \quad L_2 = \{s = "M" : P(L(M)) = \text{true}\}$$

Either  $P(\emptyset) = \text{true}$  or  $P(\emptyset) = \text{false}$ . Assume it is false (a matching proof exists if it is true). Since P is nontrivial, there is some language  $L_P$  such that  $P(L_P)$  is true. Let  $M_P$  be some Turing machine that semidecides  $L_P$ .

Let  $\tau$  construct "M\*", which operates as follows:

1. Copy its input y to another track for later.
2. Write w on its input tape and execute M on w.
3. If M halts, put y back on the tape and execute  $M_P$ .
4. If  $M_P$  halts on y, accept.

Claim: If  $M_2$  exists, then  $M_1 = M_2(M_\tau(s))$  decides  $L_1$ .

## Why?

Two cases to consider:

- " $M$ " " $w$ "  $\in H \Rightarrow M$  halts on  $w \Rightarrow M^*$  will halt on all strings that are accepted by  $M_P \Rightarrow L(M^*) = L(M_P) = L_P \Rightarrow P(L(M^*)) = P(L_P) = \text{true} \Rightarrow M_2$  decides  $P$ , so  $M_2$  accepts " $M^*$ "  $\Rightarrow M_1$  accepts.
- " $M$ " " $w$ "  $\notin H \Rightarrow M$  doesn't halt on  $w \Rightarrow M^*$  will halt on nothing  $\Rightarrow L(M^*) = \emptyset \Rightarrow P(L(M^*)) = P(\emptyset) = \text{false} \Rightarrow M_2$  decides  $P$ , so  $M_2$  rejects " $M^*$ "  $\Rightarrow M_1$  rejects.

## Using Rice's Theorem

**Theorem:** No nontrivial property of the recursively enumerable languages is decidable.

To use Rice's Theorem to show that a language  $L$  is not recursive we must:

- Specify a language property,  $P(L)$
- Show that the domain of  $P$  is the set of recursively enumerable languages.
- Show that  $P$  is nontrivial:
  - $P$  is true of at least one language
  - $P$  is false of at least one language

## Using Rice's Theorem: An Example

$L = \{s = "M" : \text{there exists a string on which Turing machine } M \text{ halts}\}.$   
 $= \{s = "M" : L(M) \neq \emptyset\}$

- Specify a language property,  $P(L)$ :  
 $P(L) = \text{True}$  iff  $L \neq \emptyset$
- Show that the domain of  $P$  is the set of recursively enumerable languages.  
The domain of  $P$  is the set of languages semidecided by some TM. This is exactly the set of RE languages.
- Show that  $P$  is nontrivial:  
 $P$  is true of at least one language:  $P(\{\epsilon\}) = \text{True}$   
 $P$  is false of at least one language:  $P(\emptyset) = \text{False}$

## Inappropriate Uses of Rice's Theorem

**Example 1:**

$L = \{s = "M" : M \text{ writes a 1 within three moves}\}.$

- Specify a language property,  $P(L)$   
 $P(M?) = \text{True}$  if  $M$  writes a 1 within three moves,  
False otherwise
- Show that the domain of  $P$  is the set of recursively enumerable languages.  
??? The domain of  $P$  is the set of all TMs, not their languages

**Example 2:**

$L = \{s = "M1" "M2" : L(M1) = L(M2)\}.$

- Specify a language property.  $P(L)$   
 $P(M1?, M2?) = \text{True}$  if  $L(M1) = L(M2)$   
False otherwise
- Show that the domain of  $P$  is the set of recursively enumerable languages.  
??? The domain of  $P$  is  $\text{RE} \times \text{RE}$

**Given a Turing Machine M, is L(M) Regular (or Context Free or Recursive)?**

Is this problem decidable?

No, by Rice's Theorem, since being regular (or context free or recursive) is a nontrivial property of the recursively enumerable languages.

We can also show this directly (via the same technique we used to prove the more general claim contained in Rice's Theorem):

**Given a Turing Machine M, is L(M) Regular (or Context Free or Recursive)?**

$$L_1 = H = \{s = "M" "w" : \text{Turing machine } M \text{ halts on input string } w\}$$

$\Downarrow \tau$

$$(?M_2) \quad L_2 = \{s = "M" : L(M) \text{ is regular}\}$$

Let  $\tau$  be the function that, from "M" and "w", constructs "M\*", whose own input is a string

$$t = "M*" "w*"$$

$M^*( "M*" "w*" )$  operates as follows:

1. Copy its input to another track for later.
2. Write w on its input tape and execute M on w.
3. If M halts, invoke U on "M\*" "w\*".
4. If U halts, halt and accept.

If  $M_2$  exists, then  $\neg M_2(M^*(s))$  decides  $L_1$  (H).

**Why?**

If M does not halt on w, then  $M^*$  accepts  $\emptyset$  (which is regular).

If M does halt on w, then  $M^*$  accepts H (which is not regular).

**Undecidable Problems About Unrestricted Grammars**

- Given a grammar G and a string w, is  $w \in L(G)$ ?
- Given a grammar G, is  $\epsilon \in L(G)$ ?
- Given two grammars  $G_1$  and  $G_2$ , is  $L(G_1) = L(G_2)$ ?
- Given a grammar G, is  $L(G) = \emptyset$ ?

**Given a Grammar G and a String w, Is  $w \in L(G)$ ?**

$$L_1 = H = \{s = "M" "w" : \text{Turing machine } M \text{ halts on input string } w\}$$

$\Downarrow \tau$

$$(?M_2) \quad L_2 = \{s = "G" "w" : w \in L(G)\}$$

Let  $\tau$  be the construction that builds a grammar G for the language L that is semidecided by M. Thus  $w \in L(G)$  iff M(w) halts.

Then  $\tau("M" "w") = "G" "w"$

If  $M_2$  exists, then  $M_1 = M_2(M_\tau(s))$  decides  $L_1$ .

## Undecidable Problems About Context-Free Grammars

- Given a context-free grammar  $G$ , is  $L(G) = \Sigma^*$ ?
- Given two context-free grammars  $G_1$  and  $G_2$ , is  $L(G_1) = L(G_2)$ ?
- Given two context-free grammars  $G_1$  and  $G_2$ , is  $L(G_1) \cap L(G_2) = \emptyset$ ?
- Is context-free grammar,  $G$  ambiguous?
- Given two pushdown automata  $M_1$  and  $M_2$ , do they accept precisely the same language?
- Given a pushdown automaton  $M$ , find an equivalent pushdown automaton with as few states as possible.

### Given Two Context-Free Grammars $G_1$ and $G_2$ , Is $L(G_1) = L(G_2)$ ?

$$L_1 = \{s = "G" \mid \text{a CFG } G \text{ and } L(G) = \Sigma^*\}$$

$$\Downarrow \qquad \tau$$

$$(?M_2) \quad L_2 = \{s = "G_1" "G_2" : G_1 \text{ and } G_2 \text{ are CFGs and } L(G_1) = L(G_2)\}$$

Let  $\tau$  append the description of a context free grammar  $G_{\Sigma^*}$  that generates  $\Sigma^*$ .

Then,  $\tau("G") = "G" "G_{\Sigma^*}"$

If  $M_2$  exists, then  $M_1 = M_2(M_\tau(s))$  decides  $L_1$ .

## Non-RE Languages

There are an uncountable number of non-RE languages, but only a countably infinite number of TM's (hence RE languages).  
 $\therefore$  The class of non-RE languages is much bigger than that of RE languages!

**Intuition:** Non-RE languages usually involve either infinite search or knowing a TM will infinite loop to accept a string.

- { $\langle M \rangle$ :  $M$  is a TM that does not halt on the empty tape}
- { $\langle M \rangle$ :  $M$  is a TM and  $L(M) = \Sigma^*$ }
- { $\langle M \rangle$ :  $M$  is a TM and there does not exist a string on which  $M$  halts }

### Proving Languages are not RE

- Diagonalization
- Complement RE, not recursive
- Reduction from a non-RE language
- Rice's theorem for non-RE languages (not covered)

### Diagonalization

$L = \{\langle M \rangle : M \text{ is a TM and } M(\langle M \rangle) \text{ does not halt}\}$  is not RE

Suppose  $L$  is RE. There is a TM  $M^*$  that semidecides  $L$ . Is  $\langle M^* \rangle$  in  $L$ ?

- If it is, then  $M^*(\langle M^* \rangle)$  halts (by the definition of  $M^*$  as a semideciding machine for  $L$ )
  - But, by the definition of  $L$ , if  $\langle M^* \rangle \in L$ , then  $M^*(\langle M^* \rangle)$  does not halt.
- Contradiction. So  $L$  is not RE.

(This is a very "bare-bones" diagonalization proof.)

Diagonalization can only be easily applied to a few non-RE languages.

## Complement of an RE, but not Recursive Language

Example:  $\bar{H} = \{ \langle M, w \rangle : M \text{ does not accept } w \}$

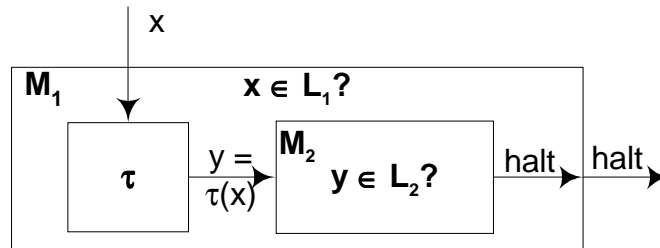
Consider  $H = \{ \langle M, w \rangle : M \text{ is a TM that accepts } w \}$ :

- $H$  is RE—it is semidecided by  $U$ , the Universal Turing Machine.
- $H$  is not recursive—it is equivalent to the halting problem, which is undecidable.

From the theorem,  $\bar{H}$  is not RE.

### Reductions and RE Languages

**Theorem:** If there is a reduction from  $L_1$  to  $L_2$  and  $L_2$  is RE, then  $L_1$  is RE.



**Theorem:** If there is a reduction from  $L_1$  to  $L_2$  and  $L_1$  is not RE, then  $L_2$  is not RE.

### Reduction from a known non-RE Language

Using a reduction from a non-RE language:

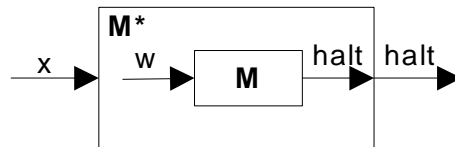
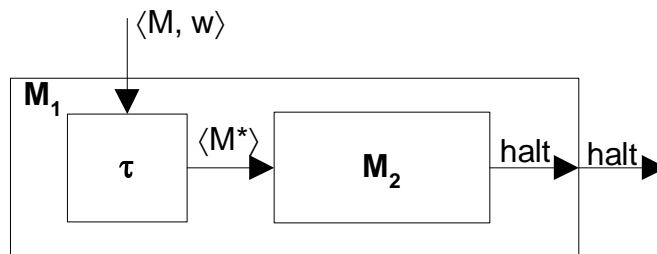
$$L_1 = \bar{H} = \{ \langle M, w \rangle : \text{Turing machine } M \text{ does not halt on input string } w \}$$

$\Downarrow \tau$

$$(?M_2) \quad L_2 = \{ \langle M \rangle : \text{there does not exist a string on which Turing machine } M \text{ halts} \}$$

Let  $\tau$  be the function that, from  $\langle M \rangle$  and  $\langle w \rangle$ , constructs  $\langle M^* \rangle$ , which operates as follows:

1. Erase the input tape ( $M^*$  ignores its input).
2. Write  $w$  on the tape
3. Run  $M$  on  $w$ .

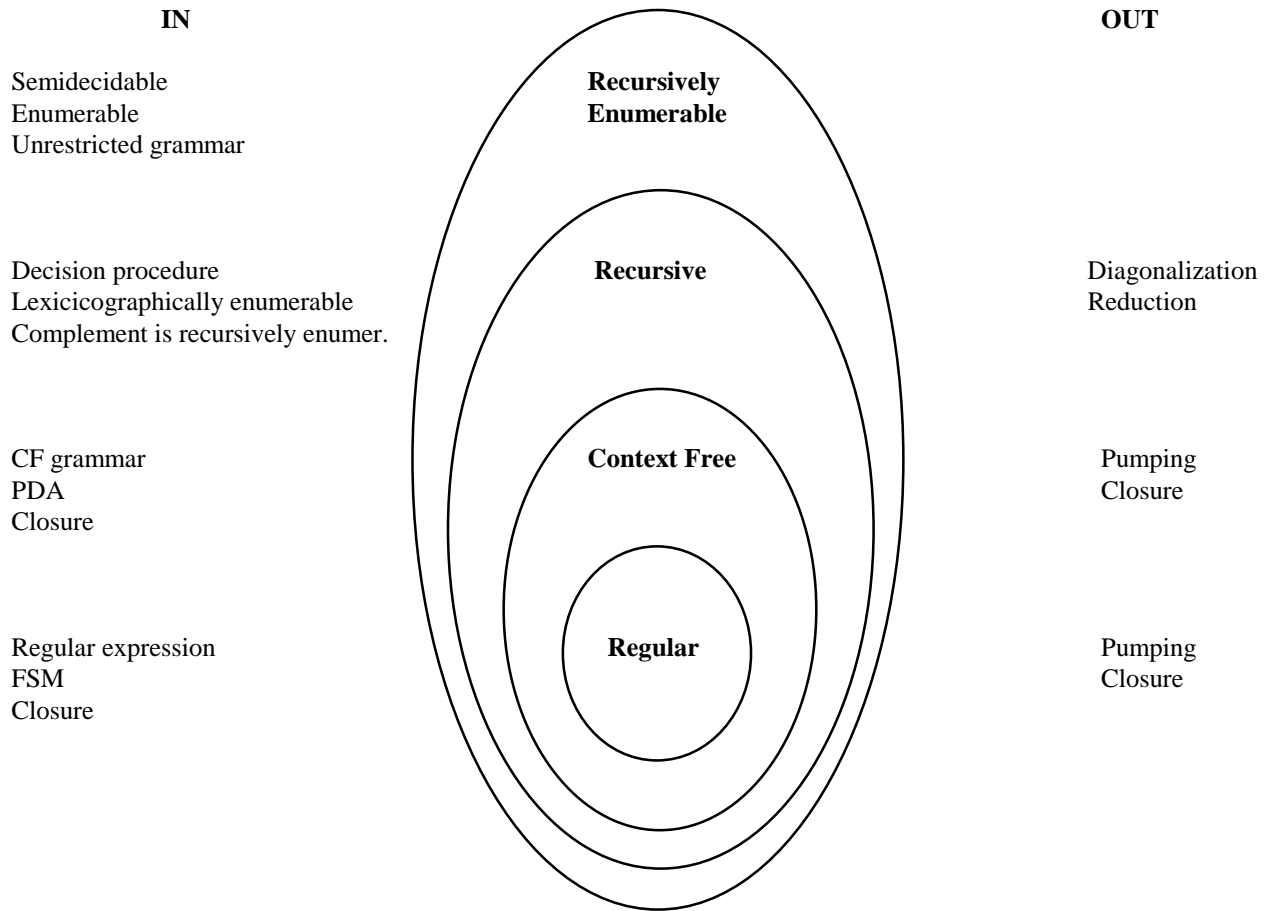


$\langle M, w \rangle \in \bar{H} \Rightarrow M$  does not halt on  $w \Rightarrow M^*$  does not halt on any input  $\Rightarrow M^*$  halts on nothing  $\Rightarrow M_2$  accepts (halts).

$\langle M, w \rangle \notin \bar{H} \Rightarrow M$  halts on  $w \Rightarrow M^*$  halts on everything  $\Rightarrow M_2$  loops.

If  $M_2$  exists, then  $M_1(\langle M, w \rangle) = M_2(M_\tau(\langle M, w \rangle))$  and  $M_1$  semidecides  $L_1$ . Contradiction.  $L_1$  is not RE.  $\therefore L_2$  is not RE.

# Language Summary



# Introduction to Complexity Theory

Read K & S Chapter 6.

Most computational problems you will face your life are solvable (decidable). We have yet to address whether a problem is “easy” or “hard”. Complexity theory tries to answer this question.

Recall that a computational problem can be recast as a language recognition problem.

Some “easy” problems:

- Pattern matching
- Parsing
- Database operations (select, join, etc.)
- Sorting

Some “hard” problems:

- Traveling salesman problem
- Boolean satisfiability
- Knapsack problem
- Optimal flight scheduling

“Hard” problems usually involve the examination of a large search space.

## Big-O Notation

- Gives a quick-and-dirty measure of function size
- Used for time and space metrics

A function  $f(n)$  is  $O(g(n))$  whenever there exists a constant  $c$ , such that  $|f(n)| \leq c \cdot |g(n)|$  for all  $n \geq 0$ .

(We are usually most interested in the “smallest” and “simplest” function,  $g$ .)

Examples:

$$2n^3 + 3n^2 \cdot \log(n) + 75n^2 + 7n + 2000 \text{ is } \underline{O(n^3)}$$

$$75 \cdot 2^n + 200n^5 + 10000 \text{ is } \underline{O(2^n)}$$

A function  $f(n)$  is *polynomial* if  $f(n)$  is  $O(p(n))$  for some polynomial function  $p$ .

If a function  $f(n)$  is not polynomial, it is considered to be *exponential*, whether or not it is  $O$  of some exponential function (e.g.  $n^{\log n}$ ).

In the above two examples, the first is polynomial and the second is exponential.

## Comparison of Time Complexities

Speed of various time complexities for different values of  $n$ , taken to be a measure of *problem size*. (Assumes 1 step per microsecond.)

$f(n) \backslash n$	10	20	30	40	50	60
$n$	.00001 sec.	.00002 sec.	.00003 sec.	.00004 sec.	.00005 sec.	.00006 sec.
$n^2$	.0001 sec.	.0004 sec.	.0009 sec.	.0016 sec.	.0025 sec.	.0036 sec.
$n^3$	.001 sec.	.008 sec.	.027 sec.	.064 sec.	.125 sec.	.216 sec.
$n^5$	.1 sec.	3.2 sec.	24.3 sec.	1.7 min.	5.2 min.	13.0 min.
$2^n$	.001 sec.	1.0 sec.	17.9 min.	12.7 days	35.7 yr.	366 cent.
$3^n$	.059 sec.	58 min.	6.5 yr.	3855 cent.	$2 \times 10^8$ cent.	$1.3 \times 10^{13}$ cent.

Faster computers don’t really help. Even taking into account Moore’s Law, algorithms with exponential time complexity are considered *intractable*. ∴ Polynomial time complexities are strongly desired.

## Polynomial Land

If  $f_1(n)$  and  $f_2(n)$  are polynomials, then so are:

- $f_1(n) + f_2(n)$
- $f_1(n) \cdot f_2(n)$
- $f_1(f_2(n))$

This means that we can sequence and compose polynomial-time algorithms with the resulting algorithms remaining polynomial-time.

## Computational Model

For formally describing the time (and space) complexities of algorithms, we will use our old friend, the deciding TM (decision procedure).

There are two parts:

- The problem to be solved must be translated into an equivalent language recognition problem.
- A TM to solve the language recognition problem takes an encoded instance of the problem (of size  $n$  symbols) as input and decides the instance in at most  $T_M(n)$  steps.

We will classify the time complexity of an algorithm (TM) to solve it by its big-O bound on  $T_M(n)$ .

We are most interested in polynomial time complexity algorithms for various types of problems.

## Encoding a Problem

**Traveling Salesman Problem:** Given a set of cities and the distances between them, what is the minimum distance tour a salesman can make that covers all cities and returns him to his starting city?

Stated as a decision question over graphs: Given a graph  $G = (V, E)$ , a positive distance function for each edge  $d: E \rightarrow \mathbb{N}^+$ , and a bound  $B$ , is there a circuit that covers all  $V$  where  $\sum d(e) \leq B$ ? (Here a *minimization* problem was turned into a *bound* problem.)

A possible encoding the problem:

- Give  $|V|$  as an integer.
- Give  $B$  as an integer.
- Enumerate all  $(v_1, v_2, d)$  as a list of triplets of integers (this gives both  $E$  and  $d$ ).
- All integers are expressed as Boolean numbers.
- Separate these entries with commas.

Note that the sizes of most “reasonable” problem encodings are polynomially related.

## What about Turing Machine Extensions?

Most TM extensions are can be simulated by a standard TM in a time polynomially related to the time of the extended machine.

- $k$ -tape TM can be simulated in  $O(T^2(n))$
- Random Access Machine can be simulated in  $O(T^3(n))$

(Real programming languages can be polynomially related to the RAM.)

BUT... The nondeterminism TM extension is different.

A nondeterministic TM can be simulated by a standard TM in  $O(2^{p(n)})$  for some polynomial  $p(n)$ .

Some faster simulation method might be possible, but we don't know it.

Recall that a nondeterministic TM can use a “guess and test” approach, which is computationally efficient at the expense of many parallel instances.



## The Class P

$P = \{ L : \text{there is a polynomial-time } \underline{\text{deterministic}} \text{ TM, } M \text{ that decides } L \}$

Roughly speaking, P is the class of problems that can be solved by deterministic algorithms in a time that is polynomially related to the size of the respective problem instance.

The way the problem is encoded or the computational abilities of the machine carrying out the algorithm are not very important.

Example: Given an integer  $n$ , is there a positive integer  $m$ , such that  $n = 4m$ ?

Problems in P are considered tractable or “easy”.

## The Class NP

$NP = \{ L : \text{there is a polynomial time } \underline{\text{nondeterministic}} \text{ TM, } M \text{ that decides } L \}$

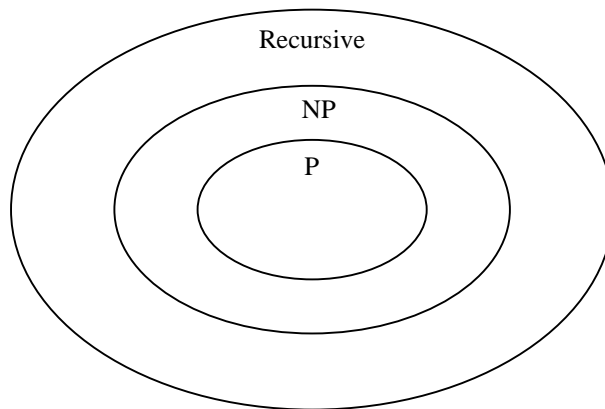
Roughly speaking, NP is the class of problems that can be solved by nondeterministic algorithms in a time that is polynomially related to the size of the respective problem instance.

Many problems in NP are considered “intractable” or “hard”.

Examples:

- **Traveling salesman problem:** Given a graph  $G = (V, E)$ , a positive distance function for each edge  $d: E \rightarrow \mathbb{N}^+$ , and a bound  $B$ , is there a circuit that covers all  $V$  where  $\sum d(e) \leq B$ ?
- **Subgraph isomorphism problem:** Given two graphs  $G_1$  and  $G_2$ , does  $G_1$  contain a subgraph isomorphic to  $G_2$ ?

## The Relationship of P and NP



We're considering only solvable (decidable) problems.

Clearly  $P \subseteq NP$ .

P is closed under complement.

NP probably isn't closed under complement. Why?

***Whether  $P = NP$  is considered computer science's greatest unsolved problem.***

## Why NP is so Interesting

- To date, nearly all decidable problems with polynomial bounds on the size of the solution are in this class.
- Most NP problems have simple nondeterministic solutions.
- The hardest problems in NP have exponential deterministic time complexities.
- Nondeterminism doesn't influence decidability, so maybe it shouldn't have a big impact on complexity.
- Showing that  $P = NP$  would dramatically change the computational power of our algorithms.

### Stephen Cook's Contribution (1971)

- Emphasized the importance of polynomial time reducibility.
- Pointed out the importance of NP.
- Showed that the Boolean Satisfiability (SAT) problem has the property that every other NP problem can be polynomially reduced to it. Thus, SAT can be considered the hardest problem in NP.
- Suggested that other NP problems may also be among the "hardest problems in NP".

This "hardest problems in NP" class is called the class of "NP-complete" problems.

Further, if any of these NP-complete problems can be solved in deterministic polynomial time, they all can and, by implication,  $P = NP$ .

Nearly all of complexity theory relies on the assumption that  $P \neq NP$ .

### Polynomial Time Reducibility

A language  $L_1$  is *polynomial time reducible* to  $L_2$  if there is a polynomial-time recursive function  $\tau$  such that  $\forall x \in L_1$  iff  $\tau(x) \in L_2$ .

If  $L_1$  is polynomial time reducible to  $L_2$ , we say  $L_1$  reduces to  $L_2$  ("polynomial time" is assumed) and we write it as  $L_1 \leq L_2$ .

**Lemma:** If  $L_1 \leq L_2$ , then  $(L_2 \in P) \Rightarrow (L_1 \in P)$ . And conversely,  $(L_1 \notin P) \Rightarrow (L_2 \notin P)$ .

**Lemma:** If  $L_1 \leq L_2$  and  $L_2 \leq L_3$  then  $L_1 \leq L_3$ .

$L_1$  and  $L_2$  are *polynomially equivalent* whenever both  $L_1 \leq L_2$  and  $L_2 \leq L_1$ .

Polynomially equivalent languages form an equivalence class. The partitions of this equivalence class are related by the partial order  $\leq$ .

$P$  is the "least" element in this partial order.

What is the "maximal" element in the partial order?

## The Class NP-Complete

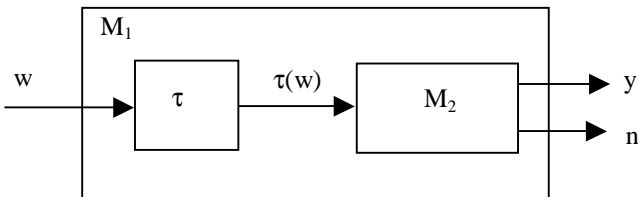
A language  $L$  is *NP-complete* if  $L \in \text{NP}$  and for all other languages  $L' \in \text{NP}$ ,  $L' \leq L$ .

NP-Complete problems are the “hardest” problems in NP.

**Lemma:** If  $L_1$  and  $L_2$  belong to NP,  $L_1$  is NP-complete and  $L_1 \leq L_2$ , then  $L_2$  is NP-complete.

Thus to prove a language  $L_2$  is NP-complete, you must do the following:

1. Show that  $L_2 \in \text{NP}$ .
2. Select a known NP-complete language  $L_1$ .
3. Construct a reduction  $\tau$  from  $L_1$  to  $L_2$ .
4. Show that  $\tau$  is polynomial-time function.



How do we get started? Is there a language that is NP-complete?

## Boolean Satisfiability (SAT)

Given a set of Boolean variables  $U = \{u_1, u_2, \dots, u_m\}$  and a Boolean expression in conjunctive normal form (conjunctions of clauses—disjunctions of variables or their negatives), is there a truth assignment to  $U$  that makes the Boolean expression true (satisfies the expression)?

Note: All Boolean expressions can be converted to conjunctive normal form.

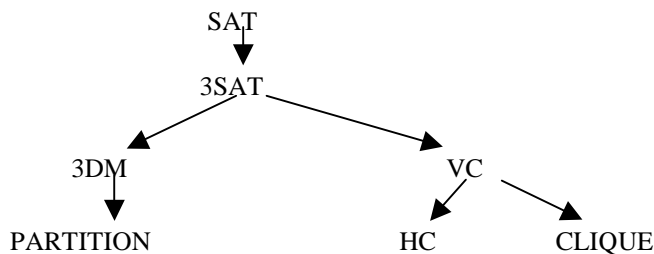
Example:  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_3 \vee x_4 \vee \neg x_2)$

**Cook’s Theorem:** SAT is NP-complete.

1. Clearly  $\text{SAT} \in \text{NP}$ .
2. The proof constructs a complex Boolean expression that satisfied exactly when a NDTM accepts an input string  $x$  where  $|w| = n$ . Because the NDTM is in NP, its running time is  $O(p(n))$ . The number of variables is polynomially related to  $p(n)$ .

**SAT is NP-complete because  $\text{SAT} \in \text{NP}$  and for all other languages  $L' \in \text{NP}$ ,  $L' \leq \text{SAT}$ .**

## Reduction Roadmap



The early NP-complete reductions took this structure. Each phrase represents a problem. The arrow represents a reduction from one problem to another.

Today, thousands of diverse problems have been shown to be NP-complete.

Let’s now look at these problems.

### 3SAT (3-satisfiability)

Boolean satisfiability where each clause has exactly 3 terms.

### 3DM (3-Dimensional Matching)

Consider a set  $M \subseteq X \times Y \times Z$  of disjoint sets,  $X$ ,  $Y$ , &  $Z$ , such that  $|X| = |Y| = |Z| = q$ . Does there exist a *matching*, a subset  $M' \subseteq M$  such that  $|M'| = q$  and  $M'$  partitions  $X$ ,  $Y$ , and  $Z$ ?

This is a generalization of the marriage problem, which has two sets men & women and a relation describing acceptable marriages. Is there a pairing that marries everyone acceptably?

The marriage problem is in P, but this “3-sex version” of the problem is NP-complete.

### PARTITION

Given a set  $A$  and a positive integer size,  $s(a) \in \mathbb{N}^+$ , for each element,  $a \in A$ . Is there a subset  $A' \subseteq A$  such that

$$\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a) \quad ?$$

### VC (Vertex Cover)

Given a graph  $G = (V, E)$  and an integer  $K$ , such that  $0 < K \leq |V|$ , is there a *vertex cover* of size  $K$  or less for  $G$ , that is, a subset  $V' \subseteq V$  such that  $|V'| \leq K$  and for each edge,  $(u, v) \in E$ , at least one of  $u$  and  $v$  belongs to  $V'$ ?

### CLIQUE

Given a graph  $G = (V, E)$  and an integer  $J$ , such that  $0 < J \leq |V|$ , does  $G$  contain a *clique* of size  $J$  or more, that is a subset  $V' \subseteq V$  such that  $|V'| \geq J$  and every two vertices in  $V'$  are joined by an edge in  $E$ ?

### HC (Hamiltonian Circuit)

Given a graph  $G = (V, E)$ , does there exist a *Hamiltonian circuit*, that is an ordering  $\langle v_1, v_2, \dots, v_n \rangle$  of all  $V$  such that  $(v_i, v_{i+1}) \in E$  and  $(v_n, v_1) \in E$  for all  $i$ ,  $1 \leq i < |V|$ ?

### Traveling Salesman Prob. is NP-complete

Given a graph  $G = (V, E)$ , a positive distance function for each edge  $d: E \rightarrow \mathbb{N}^+$ , and a bound  $B$ , is there a circuit that covers all  $V$  where  $\sum d(e) \leq B$ ?

To prove a language TSP is NP-complete, you must do the following:

1. Show that  $TSP \in NP$ .
2. Select a known NP-complete language  $L_1$ .
3. Construct a reduction  $\tau$  from  $L_1$  to TSP.
4. Show that  $\tau$  is polynomial-time function.

**TSP  $\in$  NP:** Guess a set of roads. Verify that the roads form a tour that hits all cities. Answer “yes” if the guess is a tour and the sum of the distances is  $\leq B$ .

**Reduction from HC:** Answer the Hamiltonian circuit question on  $G = (V, E)$  by constructing a complete graph where “roads” have distance 1 if the edge is in  $E$  and 2 otherwise. Pose the TSP problem, is there a tour of length  $\leq |V|$ ?

## Notes on NP-complete Proofs

The more NP-complete problems are known, the easier it is to find a NP-complete problem to reduce from.

Most reductions are somewhat complex.

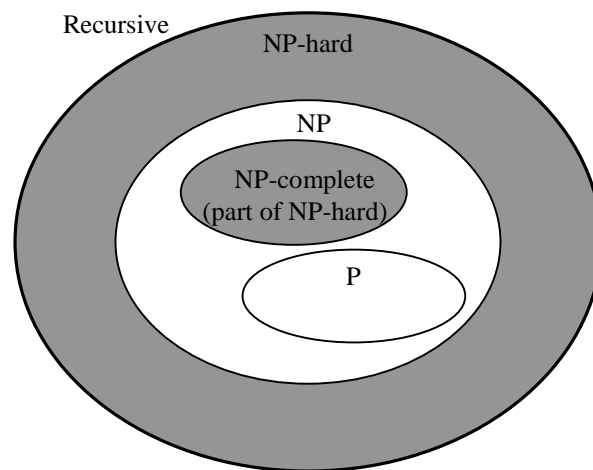
It is sufficient to show that a restricted version of the problem is NP-complete.

### More Theory

NP has a rich structure that includes more than just P and NP-complete. This structure is studied in later courses on the theory of computation.

The set of recursive problems outside of NP (and including NP-complete) are called *NP-hard*. There is a proof technique to show that such problems are at least as hard as NP-complete problems.

*Space complexity* addresses how much tape does a TM use in deciding a language. There is a rich set of theories surrounding space complexity.



### Dealing with NP-completeness

You will likely run into NP-complete problems in your career. For example, most optimization problems are NP-complete.

Some techniques for dealing with intractable problems:

- Recognize when there is a tractable special case of the general problem.
- Use other techniques to limit the search space.
- For optimization problems, seek a near-optimal solution.

The field of *linear optimization* springs out of the latter approach. Some linear optimization solutions can be proven to be “near” optimal.

A branch of complexity theory deals with solving problems within some error bound or probability.

For more: Read [Computers and Intractability: A Guide to the Theory of NP-Completeness](#) by Michael R. Garey and David S. Johnson, 1979.

## II. Homework

**CS 341 Homework 1**  
**Basic Techniques**

1. What are these sets? Write them using braces, commas, numerals, ... (for infinite sets), and  $\emptyset$  only.
  - (a)  $(\{1, 3, 5\} \cup \{3, 1\}) \cap \{3, 5, 7\}$
  - (b)  $\cup\{\{3\}, \{3, 5\}, \cap\{\{5, 7\}, \{7, 9\}\}\}$
  - (c)  $(\{1, 2, 5\} - \{5, 7, 9\}) \cup (\{5, 7, 9\} - \{1, 2, 5\})$
  - (d)  $2^{\{7, 8, 9\}} - 2^{\{7, 9\}}$
  - (e)  $2^{\emptyset}$
  - (f)  $\{x : \exists y \in \mathbb{N} \text{ where } x = y^2\}$
  - (g)  $\{x : x \text{ is an integer and } x^2 = 2\}$
  
2. Prove each of the following:
  - (a)  $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
  - (b)  $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
  - (c)  $A \cap (A \cup B) = A$
  - (d)  $A \cup (A \cap B) = A$
  - (e)  $A - (B \cap C) = (A - B) \cup (A - C)$
  
3. Write each of the following explicitly:
  - (a)  $\{1\} \times \{1, 2\} \times \{1, 2, 3\}$
  - (b)  $\emptyset \times \{1, 2\}$
  - (c)  $2^{\{1, 2\}} \times \{1, 2\}$
  
4. Let  $R = \{(a, b), (a, c), (c, d), (a, a), (b, a)\}$ . What is  $R \circ R$ , the composition of  $R$  with itself? What is  $R^{-1}$ , the inverse of  $R$ ? Is  $R$ ,  $R \circ R$ , or  $R^{-1}$  a function?
  
5. What is the cardinality of each of the following sets? Justify your answer.
  - (a)  $S = \mathbb{N} - \{2, 3, 4\}$
  - (b)  $S = \{\emptyset, \{\emptyset\}\}$
  - (c)  $S = 2^{\{a, b, c\}}$
  - (d)  $S = \{a, b, c\} \times \{1, 2, 3, 4\}$
  - (e)  $S = \{a, b, \dots, z\} \times \mathbb{N}$
  
6. Consider the chart of example relations in Section 3.2. For the first six, give an example that proves that the relation is missing each of the properties that the chart claims it is missing. For example, show that Mother-of is not reflexive, symmetric, or transitive.
  
7. Let  $A, B$  be two sets. If  $2^A = 2^B$ , must  $A = B$ ? Prove your answer.
  
8. For each of the following sets, state whether or not it is a partition of  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ .
  - (a)  $\{\{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}, \{9\}, \{10\}\}$
  - (b)  $\{\emptyset, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}, \{9\}, \{10\}\}$
  - (c)  $\{\{1, 2\}, \{3, 4\}, \{5, 6\}, \{7, 8\}, \{9, 10\}\}$
  - (d)  $\{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{4, 5\}, \{5, 6\}, \{6, 7\}, \{7, 8\}, \{8, 9\}, \{9, 10\}\}$
  
9. For each of the following relations, state whether it is a partial order (that is not also total), a total order, or neither. Justify your answer.
  - (a) DivisibleBy, defined on the natural numbers.  $(x, y) \in \text{DivisibleBy}$  iff  $x$  is evenly divisible by  $y$ . So, for example,  $(9, 3) \in \text{DivisibleBy}$  but  $(9, 4) \notin \text{DivisibleBy}$ .

(b) LessThanOrEqual defined on ordered pairs of natural numbers.  $(a, b) \leq (x, y)$  iff  $a \leq x$  or  $(a = x$  and  $b \leq y)$ . For example,  $(1,2) \leq (2,1)$  and  $(1,2) \leq (1,3)$ .

(c) The relation defined by the following boolean matrix:

1		1	
	1	1	
		1	1
1			1

10. Are the following sets closed under the following operations? If not, what are the respective closures?

- (a) The odd integers under multiplication.
- (b) The positive integers under division.
- (c) The negative integers under subtraction.
- (d) The negative integers under multiplication.
- (e) The odd length strings under concatenation.

11. What is the reflexive transitive closure  $R^*$  of the relation

$$R = \{(a, b), (a, c), (a, d), (d, c), (d, e)\}$$
 Draw a directed graph representing  $R^*$ .

12. For each of the following relations  $R$ , over some domain  $D$ , compute the reflexive, symmetric, transitive closure  $R'$ . Try to think of a simple descriptive name for the new relation  $R'$ . Since  $R'$  must be an equivalence relation, describe the partition that  $R$  induces on  $D$ .

- (a) Let  $D$  be the set of 50 states in the US.  $\forall xy, xRy$  iff  $x$  shares a boundary with  $y$ .
- (b) Let  $D$  be the natural numbers.  $\forall xy, xRy$  iff  $y = x+3$ .
- (c) Let  $D$  be the set of strings containing no symbol except  $a$ .  $\forall xy, xRy$  iff  $y = xa$ . (i.e., if  $y$  equals  $x$  concatenated with  $a$ ).

13. Consider an infinite rectangular grid (like an infinite sheet of graph paper). Let  $S$  be the set of intersection points on the grid. Let each point in  $S$  be represented as a pair of  $(x,y)$  coordinates where adjacent points differ in one coordinate by exactly 1 and coordinates increase (as is standard) as you move up and to the right.

(a) Let  $R$  be the following relation on  $S$ :  $\forall (x_1,y_1)(x_2,y_2), (x_1,y_1)R(x_2,y_2)$  iff  $x_2 = x_1+1$  and  $y_2 = y_1+1$ . Let  $R'$  be the reflexive, symmetric, transitive closure of  $R$ . Describe in English the partition  $P$  that  $R'$  induces on  $S$ . What is the cardinality of  $P$ ?

(b) Let  $R$  be the following relation on  $S$ :  $\forall (x_1,y_1)(x_2,y_2), (x_1,y_1)R(x_2,y_2)$  iff  $(x_2 = x_1+1$  and  $y_2 = y_1+1)$  or  $(x_2 = x_1-1$  and  $y_2 = y_1+1)$ . Let  $R'$  be the reflexive, symmetric, transitive closure of  $R$ . Describe in English the partition  $P$  that  $R'$  induces on  $S$ . What is the cardinality of  $P$ ?

(c) Let  $R$  be the following relation on  $S$ :  $\forall (x_1,y_1)(x_2,y_2), (x_1,y_1)R(x_2,y_2)$  iff  $(x_2,y_2)$  is reachable from  $(x_1,y_1)$  by moving two squares in any one of the four directions and then one square in a perpendicular direction. Let  $R'$  be the reflexive, symmetric, transitive closure of  $R$ . Describe in English the partition  $P$  that  $R'$  induces on  $S$ . What is the cardinality of  $P$ ?

14. Is the transitive closure of the symmetric closure of a binary relation necessarily reflexive? Prove it or give a counterexample.

15. Give an example of a binary relation that is not reflexive but has a transitive closure that is reflexive.

16. For each of the following functions, state whether or not it is (i) one-to-one, (ii) onto, and (iii) idempotent. Justify your answers.

- (a)  $+$ :  $P \times P \rightarrow P$ , where  $P$  is the set of positive integers, and  $+(a, b) = a + b$  (In other words, simply addition defined on the positive integers)
- (b)  $X$ :  $B \times B \rightarrow B$ , where  $B$  is the set  $\{\text{True}, \text{False}\}$



$X(a, b)$  = the exclusive or of  $a$  and  $b$

17. Consider the following set manipulation problems:

(a) Let  $S = \{a, b\}$ . Let  $T = \{b, c\}$ . List the elements of  $P$ , defined as  

$$P = 2^S \cap 2^T.$$

(b) Let  $Z$  be the set of integers. Let  $S = \{x \in Z: \exists y \in Z \text{ and } x = 2y\}$ . Let  $T = \{x \in Z: \exists y \in Z \text{ and } x = 2^y\}$ . Let  $W = S - T$ . Describe  $W$  in English. List any five consecutive elements of  $W$ . Let  $X = T - S$ . What is  $X$ ?

### Solutions

1. (a)  $\{3, 5\}$   
 (b)  $\{3, 5, 7\}$   
 (c)  $\{1, 2, 7, 9\}$   
 (d)  $\{8\}, \{7, 8\}, \{8, 9\}, \{7, 8, 9\}$   
 (e)  $\{\emptyset\}$   
 (f)  $\{0, 1, 4, 9, 25, 36, \dots\}$  (the perfect squares)  
 (g)  $\emptyset$  (since the square root of 2 is not an integer)

2. (a)  $A \cup (B \cap C) = (B \cap C) \cup A$                       commutativity  
 $= (B \cup A) \cap (C \cup A)$                       distributivity  
 $= (A \cup B) \cap (A \cup C)$                       commutativity

(b)  $A \cap (B \cup C) = (B \cup C) \cap A$                       commutativity  
 $= (B \cap A) \cup (C \cap A)$                       distributivity  
 $= (A \cap B) \cup (A \cap C)$                       commutativity

(c)  $A \cap (A \cup B) = (A \cup B) \cap A$                       commutativity  
 $= A$                       absorption

3. (a)  $\{(1,1,1), (1,1,2), (1,1,3), (1,2,1), (1,2,2), (1,2,3)\}$   
 (b)  $\emptyset$   
 (c)  $\{(\emptyset,1), (\emptyset,2), (\{1\}, 1), (\{1\}, 2), (\{2\}, 1), (\{2\}, 2), (\{1,2\}, 1), (\{1,2\}, 2)\}$

4.  $R \circ R = \{(a, a), (a, d), (a, b), (b, b), (b, c), (b, a), (a, c)\}$   
 $R \text{ inverse} = \{(b, a), (c, a), (d, c), (a, a), (a, b)\}$   
 None of  $R, R \circ R$  or  $R \text{ inverse}$  is a function.

5. (a)  $S = \{0, 1, 5, 6, 7, \dots\}$ .  $S$  has the same number of elements as  $N$ . Why? Because there is a bijection between  $S$  and  $N$ :  $f: S \rightarrow N$ , where  $f(0) = 0, f(1) = 1, \forall x \geq 5, f(x) = x - 3$ . So  $|S| = \aleph_0$ .  
 (b) 2.  
 (c)  $S =$  all subsets of  $\{a, b, c\}$ . So  $S = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$ . So  $|S| = 8$ . We could also simply have used the fact that the cardinality of the power set of a finite set of cardinality  $c$  is  $2^c$ .  
 (d)  $S = \{(a, 1), (a, 2), (a, 3), (a, 4), (b, 1), (b, 2), (b, 3), (b, 4), (c, 1), (c, 2), (c, 3), (c, 4)\}$ . So  $|S| = 12$ . Or we could have used the fact that, for finite sets,  $|A \times B| = |A| * |B|$ .  
 (e)  $S = \{(a, 0), (a, 1), \dots, (b, 0), (b, 1), \dots\}$  Clearly  $S$  contains an infinite number of elements. But are there the same number of elements in  $S$  as in  $N$ , or are there more (26 times more, to be precise)? The answer is that there are the same number.  $|S| = \aleph_0$ . To prove this, we need a bijection from  $S$  to  $N$ . We can define this bijection as an enumeration of the elements of  $S$ :  
 $(a, 0), (b, 0), (c, 0), \dots$  (First enumerate all 26 elements of  $S$  that have 0 as their second element)

(a, 1), (b, 1), (c, 1), ... (Next enumerate all 26 elements of S that have 1 as their second element) and so forth.

- 6. Mother-of:** Not reflexive: Eve is not the mother of Eve (in fact, no one is her own mother).  
Not symmetric: mother-of(Eve, Cain), but not Mother-of(Cain, Eve).  
Not transitive: Each person has only one mother, so if Mother-of(x, y) and Mother-of(y, z), the only way to have Mother-of(x, z) would be if x and y are the same person, but we know that that's not possible since Mother-of(x, y) and no one can be the mother of herself).
- Would-recognize-picture-of:**  
Not symmetric: W-r-p-o(Elaine, Bill Clinton), but not W-r-p-o (Bill Clinton, Elaine)  
Not transitive: W r-p-o(Elaine, Bill Clinton) and W r-p-o(Bill Clinton, Bill's mom) but not W-r-p-o(Elaine, Bill's mom)
- Has-ever-been-married-to:** Not reflexive: No one is married to him or herself.  
Not transitive: H-e-b-m-t(Dave, Sue) and H-e-b-m-t(Sue, Jeff) but not H-e-b-m-t(Dave, Jeff)
- Ancestor-of:** Not reflexive: not Ancestor-of(Eve, Eve) (in fact, no one is their own ancestor).  
Not symmetric: Ancestor-of(Eve, Cain) but not Ancestor-of(Cain, Eve)
- Hangs-out-with:** Not transitive: Hangs-out-with(Bill, Monica) and Hangs-out-with(Monica, Linda Tripp), but not Hangs-out-with(Bill, Linda Tripp).
- Less-than-or-equal-to:** Not symmetric:  $1 \leq 2$ , but not  $2 \leq 1$ .

**7.** Yes, if  $2^A = 2^B$ , then A must equal B. Suppose it didn't. Then there is some element x that is in one set but not the other. Call the set x is in A. Then  $2^A$  must contain {x}, which must not be in  $2^B$ , since  $x \notin B$ . This would mean that  $2^A \neq 2^B$ , which contradicts our premise.

- 8. (a)** yes  
**(b)** no, since no element of a partition can be empty.  
**(c)** no, 0 is missing  
**(d)** no, since, each element of the original set S must appear in only one element of a partition of S.

**9. (a)** DivisibleBy is a partial order.  $\forall x (x, x) \in \text{DivisibleBy}$ , so DivisibleBy is reflexive. For x to be DivisibleBy y, x must be greater than or equal to y. So the only way for both (x, y) and (y, x) to be in DivisibleBy is for x and y to be equal. Thus DivisibleBy is antisymmetric. And if x is DivisibleBy y and y is DivisibleBy z, then x is DivisibleBy z. So DivisibleBy is transitive. But DivisibleBy is not a total order. For example neither (2, 3) nor (3, 2) is in it.

**(b)** LessThanOrEqual defined on ordered pairs is a total order. This is easy to show by relying on the fact that  $\leq$  for the natural numbers is a total order.

**(c)** This one is not a partial order at all because, although it is reflexive and antisymmetric, it is not transitive. For example, it includes (4, 1) and (1, 3), but not (4, 3).

**10. (a)** The odd integers are closed under multiplication. Every odd integer can be expressed as  $2n+1$  for some value of  $n \in \mathbb{N}$ . So the product of any two odd integers can be written as  $(2n+1)(2m+1)$  for some values of n and m. Multiplying this out, we get  $4(n+m) + 2n + 2m + 1$ , which we can rewrite as  $2(2(n+m) + n + m) + 1$ , which must be odd.

**(b)** The positive integers are not closed under division. To show that a set is not closed under an operation, it is sufficient to give one counterexample.  $1/2$  is not an integer. The closure of the positive integers under division is the positive rationals.

**(c)** The negative integers are not closed under subtraction.  $-2 - (-4) = 2$ . The closure of the negative numbers under subtraction is the integers.

**(d)** The negative integers are not closed under multiplication.  $-2 * -2 = 4$ . The closure of the negative numbers under multiplication is the nonzero integers. Remember that the closure is the *smallest* set that

contains all the necessary elements. Since it is not possible to derive zero by multiplying two negative numbers, it must not be in the closure set.

(e) The odd length strings are not closed under concatenation. "a" || "b" = "ab", which is of length 2. The closure is the set of strings of length  $\geq 2$ . Note that strings of length 1 are not included. Why?

11.  $R^* = R \cup \{(x, x) : x \in \{a, b, c, d, e\}\} \cup \{(a, e)\}$

12. (a) The easiest way to start to solve a problem like this is to start writing down the elements of  $R'$  and see if a pattern emerges. So we start with the elements of  $R$ :  $\{(TX, LA), (LA, TX), (TX, NM), (NM, TX), (LA, Ark), (Ark, LA), (LA, Miss), (Miss, LA) \dots\}$ . To construct  $R'$ , we first add all elements of the form  $(x, x)$ , so we add  $(TX, TX)$ , and so forth. Then we add the elements required to establish transitivity:

- $(NM, TX), (TX, LA) \Rightarrow (NM, TX)$
- $(TX, LA), (LA, Ark) \Rightarrow (TX, Ark)$
- $(NM, TX), (TX, Ark) \Rightarrow (NM, Ark)$ , and so forth.

If we continue this process, we will see that the reflexive, symmetric, transitive closure  $R'$  relates all states except Alaska and Hawaii to each other and each of them only to themselves. So  $R'$  can be described as relating two states if it's possible to drive from one to the other without leaving the country. The partition is:

- [Alaska]
- [Hawaii]
- [all other 48 states]

(b)  $R$  includes, for example  $\{(0, 3), (3, 6), (6, 9), (9, 12) \dots\}$ . When we compute the transitive closure, we add, among other things  $\{(0, 6), (0, 9), (0, 12)\}$ . Now try this starting with  $(1, 4)$  and  $(2, 5)$ . It's clear that  $\forall x, y, xR'y$  iff  $x = y \pmod 3$ . In other words, two numbers are related iff they have the same remainder mod 3. The partition is:

- [0, 3, 6, 9, 12 ...]
- [1, 4, 7, 10, 13 ...]
- [2, 5, 8, 11, 14 ...]

(c)  $R'$  relates all strings composed solely of a's to each other. So the partition is

- [ $\epsilon, a, aa, aaa, aaaa, \dots$ ]

13. (a) Think of two points being related via  $R$  if you can get to the second one by starting at the first and moving up one square and right one square. When we add transitivity, we gain the ability to move diagonally by two squares, or three, or whatever. So  $P$  is an infinite set. Each element of  $P$  consists of the set of points that fall on an infinite diagonal line running from lower left to upper right.

(b) Now we can move upward on either diagonal. And we can move up and right followed by up and left, and so forth. The one thing we can't do is move directly up or down or right or left exactly one square. So take any given point. To visualize the points to which it is related under  $R'$ , imagine a black and white chess board where the squares correspond to points on our grid. Each point is related to all other points of the same color. Thus the cardinality of  $P$  is 2.

(c) Now every point is related to every other point. The cardinality of  $P$  is 1.

14. You might think that for all relations  $R$  on some domain  $D$ , the transitive closure of the symmetric closure of  $R$  (call it  $TC(SC(R))$ ) must be reflexive because for any two elements  $x, y \in D$  such that  $(x, y) \in R$ , we'll have  $(x, y), (y, z) \in SC(R)$  and therefore  $(x, x), (y, y) \in TC(SC(R))$ . This is all true, but does not prove that for all  $z \in D, (z, z) \in TC(SC(R))$ . Why not? Suppose there is a  $z \in D$  such that there is no  $y \in D$  for which  $(y, z) \in R$  or  $(z, y) \in R$ . (If you look at the graph of  $R$ ,  $z$  is an isolated vertex with no edges in or out.) Then  $(z, z) \notin TC(SC(R))$ . So the answer is no, with  $R = \emptyset$  on domain  $\{a\}$  as a simple counterexample:  $TC(SC(R)) = \emptyset$ , yet it should contain  $(a, a)$  if it were reflexive.

15.  $R = \{(a, b), (b, a)\}$  on domain  $\{a, b\}$  does the trick easily.

- 16. (a)** (i)  $+$  is not one-to-one. For example  $+(1, 3) = +(2, 2) = 4$ .  
 (ii)  $+$  is not onto. There are no two positive integers that sum to 1.  
 (iii)  $+$  is not idempotent.  $+(1, 1) \neq 1$ .
- (b)** (i)  $X$  is not one-to-one. For example  $\text{True} X \text{True} = \text{False} X \text{False} = \text{False}$ .  
 (ii)  $X$  is onto. Proof:  $\text{True} X \text{True} = \text{False}$ .  $\text{True} X \text{False} = \text{True}$ . In general, when the domain is a finite set, it's easy to show that a function is onto: just show one way to derive each element.  
 (iii)  $X$  is not idempotent.  $\text{True} X \text{True} = \text{False}$ .

**17. (a)**  $P = \{\emptyset, \{b\}\}$

**(b)**  $S$  is the set of even numbers.  $T$  is the set powers of 2.  $W$  is the set of even numbers that are not powers of 2. So  $W = \{ \dots -6, -4, -2, 0, 6, 10, 12, 14, 18, \dots \}$ .  $X$  is the set of numbers that are powers of 2 but are not even. There's only one element of  $X$ .  $X = \{1\}$ .

## CS 341 Homework 2 Strings and Languages

1. Let  $\Sigma = \{a, b\}$ . Let  $L_1 = \{x \in \Sigma^* : |x| < 4\}$ . Let  $L_2 = \{aa, aaa, aaaa\}$ . List the elements in each of the following languages  $L$ :

- (a)  $L_3 = L_1 \cup L_2$
- (b)  $L_4 = L_1 \cap L_2$
- (c)  $L_5 = L_1 L_2$
- (d)  $L_6 = L_1 - L_2$

2. Consider the language  $L = a^n b^n c^m$ . Which of the following strings are in  $L$ ?

- (a)  $\epsilon$       (b)  $ab$       (c)  $c$       (d)  $aabc$       (e)  $aabbcc$       (f)  $abcc$

3. It probably seems obvious to you that if you reverse a string, the character that was originally first becomes last. But the definition we've given doesn't say that; it says only that the character that was originally last becomes first. If we want to be able to use our intuition about what happens to the first character in a proof, we need to turn it into a theorem. Prove  $\forall x, a$  where  $x$  is a string and  $a$  is a single character,  $(ax)^R = x^R a$ .

4. For each of the following binary functions, state whether or not it is (i) one-to-one, (ii) onto, (iii) idempotent, (iv) commutative, and (v) associative. Also (vi) state whether or not it has an identity, and, if so, what it is. Justify your answers.

- (a)  $\parallel : S \times S \rightarrow S$ , where  $S$  is the set of strings of length  $\geq 0$   
 $\parallel(a, b) = a \parallel b$  (In other words, simply concatenation defined on strings)
- (b)  $\parallel : L \times L \rightarrow L$  where  $L$  is a language over some alphabet  $\Sigma$   
 $\parallel(a, b) = \{w \in \Sigma^* : w = x \parallel y \text{ for some } x \in a \text{ and } y \in b\}$  In other words, the concatenation of two languages  $A$  and  $B$  is the set of strings that can be derived by taking a string from  $A$  and then concatenating onto it a string from  $B$ .

5. We can define a unary function  $F$  to be *self-inverse* iff  $\forall x \in \text{Domain}(F) F(F(x)) = x$ . The Reverse function on strings is self-inverse, for example.

- (a) Give an example of a self-inverse function on the natural numbers, on sets, and on booleans.
- (b) Prove that the Reverse function on strings is self-inverse.

### Solutions

1. First we observe that  $L_1 = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb\}$ .

- (a)  $L_3 = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, aaaa\}$
- (b)  $L_4 = \{aa, aaa\}$

(c)  $L_5 =$  every way of selecting one element from  $L_1$  followed by one element from  $L_2$ :  
 $\{\epsilon aa, aaa, baa, aaaa, abaa, baaa, bbaa, aaaaa, aabaa, abaaa, abbaa, baaaa, babaa, bbbaa, bbbbaa\} \cup$   
 $\{\epsilon aaa, aaaa, baaa, aaaaa, abaaa, baaaa, bbaaa, aaaaaa, aabaaa, abaaaa, abbaaa, baaaaa, babaaa, bbbaaa, bbbbaa\}$ . Note that we've written  $\epsilon aa$ , just to make it clear how this string was derived. It should actually be written as just  $aa$ . Also note that some elements are in both of these sets (i.e.,

there's

more than one way to derive them). Eliminating duplicates (since  $L$  is a set and thus does not contain duplicates), we get:

$\{aa, aaa, baa, aaaa, abaa, baaa, bbaa, aaaaa, aabaa, abaaa, abbaa, baaaa, babaa, bbbaa, aaaaaa, aabaaa, abaaaa, abbaaa, baaaaa, babaaa, bbbaaa, bbbbaa\}$

- (d)  $L_6 =$  every string that is in  $L_1$  but not in  $L_2$ :  $\{\epsilon, a, b, ab, ba, bb, aab, aba, abb, baa, bab, bba, bbb\}$ .

2. (a) Yes.  $n = 0$  and  $m = 0$ .  
 (b) Yes.  $n = 1$  and  $m = 0$ .  
 (c) Yes.  $n = 0$  and  $m = 1$ .  
 (d) No. There must be equal numbers of a's and b's.  
 (e) Yes.  $n = 2$  and  $m = 2$ .  
 (f) No. There must be equal numbers of a's and b's.

3. Prove:  $\forall x, a$  where  $x$  is a string and  $a$  is a single character,  $(ax)^R = x^R a$ . We'll use induction on the length of  $x$ . If  $|x| = 0$  (i.e.  $x = \epsilon$ ), then  $(a\epsilon)^R = a = \epsilon^R a$ . Next we show that if this is true for all strings of length  $n$ , then it is true for all strings of length  $n + 1$ . Consider any string  $x$  of length  $n + 1$ . Since  $|x| > 0$ , we can rewrite  $x$  as  $y b$  for some single character  $b$ .

$$\begin{aligned}
 (ax)^R &= (ayb)^R && \text{Rewrite of } x \text{ as } yb \\
 &= b(ay)^R && \text{Definition of reversal} \\
 &= b(y^R a) && \text{Induction hypothesis (since } |x| = n + 1, |y| = n) \\
 &= (b y^R) a && \text{Associativity of concatenation} \\
 &= x^R a && \text{Definition of reversal: If } x = yb \text{ then } x^R = by^R
 \end{aligned}$$

4. (a) (i)  $\parallel$  is not one-to-one. For example,  $\parallel(ab, c) = \parallel(a, bc) = abc$ .  
 (ii)  $\parallel$  is onto. Proof:  $\forall s \in S, \parallel(s, \epsilon) = s$ , so every element of  $s$  can be generated.  
 (iii)  $\parallel$  is not idempotent.  $\parallel(a, a) \neq a$ .  
 (iv)  $\parallel$  is not commutative.  $\parallel(ab, cd) \neq \parallel(cd, ab)$   
 (v)  $\parallel$  is associative.  
 (vi)  $\parallel$  has  $\epsilon$  as both a left and right identity.  
 (b) (i)  $\parallel$  is not one to one. For example, Let  $\Sigma = \{a, b, c\}$ .  $\parallel(\{a\}, \{bc\}) = \{abc\} = \parallel(\{ab\}, \{c\})$   
 (ii)  $\parallel$  is onto. Proof:  $\forall L \subseteq \Sigma^*, \parallel(L, \{\epsilon\}) = L$ , so every element of  $s$  can be generated. Notice that this proof is very similar to the one we used to show that concatenation of strings is onto. Both proofs rely

on

the fact that  $\epsilon$  is an identity for concatenation of strings. Given the way in which we defined concatenation of languages as the concatenation of strings drawn from the two languages,  $\{\epsilon\}$  is an identity for concatenation of languages and thus it enables us to prove that all languages can be derived from the concatenation operation.

- (iii)  $\parallel$  is not idempotent.  $\parallel(\{a\}, \{a\}) = \{aa\}$   
 (iv)  $\parallel$  is not commutative.  $\parallel(\{a\}, \{b\}) = \{ab\}$ . But  $\parallel(\{b\}, \{a\}) = \{ba\}$ .  
 (v)  $\parallel$  is associative.  
 (vi)  $\parallel$  has  $\{\epsilon\}$  as both a left and right identity.

5. (a) Integers:  $F(x) = -x$  is self-inverse. Sets: Complement is self-inverse. Booleans: Not is self-inverse.  
 (b) We'll prove this by induction on the length of the string.

Base case: If  $|x| = 0$  or  $1$ , then  $x^R = x$ . So  $(x^R)^R = x^R = x$ .

Show that if this is true for all strings of length  $n$ , then it is true for all strings of length  $n + 1$ . Any string  $s$  of length  $n + 1$  can be rewritten as  $xa$  for some single character  $a$ . So now we have:

$$\begin{aligned}
 s^R &= a x^R && \text{definition of string reversal} \\
 (s^R)^R &= (a x^R)^R && \text{substituting a } x^R \text{ for } s^R \\
 &= (x^R)^R a && \text{by the theorem we proved above in (3)} \\
 &= xa && \text{induction hypothesis} \\
 &= s && \text{since } xa \text{ was just a way of rewriting } s
 \end{aligned}$$

**CS 341 Homework 3**  
**Languages and Regular Expressions**

1. Describe in English, as briefly as possible, each of the following (in other words, describe the language defined by each regular expression):

- (a)  $L((a^*a)b \cup b)$   
 (b)  $L(((a^*b^*)^*ab) \cup ((a^*b^*)^*ba))(b \cup a)^*$

2. Rewrite each of these regular expressions as a simpler expression representing the same set.

- (a)  $\emptyset^* \cup a^* \cup b^* \cup (a \cup b)^*$   
 (b)  $((a^*b^*)^* (b^*a^*)^*)^*$   
 (c)  $(a^*b)^* \cup (b^*a)^*$

3. Let  $\Sigma = \{a, b\}$ . Write regular expressions for the following sets:

- (a) All strings in  $\Sigma^*$  whose number of a's is divisible by three.  
 (b) All strings in  $\Sigma^*$  with no more than three a's.  
 (c) All strings in  $\Sigma^*$  with exactly one occurrence of the substring aaa.

4. Which of the following are true? Prove your answer.

- (a)  $baa \in a^*b^*a^*b^*$   
 (b)  $b^*a^* \cap a^*b^* = a^* \cup b^*$   
 (c)  $a^*b^* \cap c^*d^* = \emptyset$   
 (d)  $abcd \in (a(cd)^*b)^*$

5. Show that  $L((a \cup b)^*) = L(a^* (ba^*)^*)$ .

6. Consider the following:

- (a)  $((a \cup b) \cup (ab))^*$   
 (b)  $(a^+ a^n b^n)$   
 (c)  $((ab)^* \emptyset)$   
 (d)  $((ab \cup c)^* \cap (b \cup c^*))$   
 (e)  $(\emptyset^* \cup (bb^*))$

- (i) Which of the above are “pure” regular expressions?  
 (ii) For each of the above that is a regular expression, give a simplified equivalent “pure” regular expression.  
 (iii) Which of the above represent regular languages?

7. True - False: For all languages  $L_1, L_2,$  and  $L_3$

- (a)  $(L_1L_2)^* = L_1^*L_2^*$   
 (b)  $(L_1 \cup L_2)^* = L_1^* \cup L_2^*$   
 (c)  $(L_1 \cup L_2)L_3 = L_1L_3 \cup L_2L_3$   
 (d)  $(L_1L_2) \cup L_3 = (L_1 \cup L_3)(L_2 \cup L_3)$   
 (e)  $L_1^+)^* = L_1^*$   
 (f)  $(L_1^+)^+ = L_1^+$   
 (g)  $(L_1^*)^+ = (L_1^+)^*$   
 (h)  $L_1^* = L_1^+ \cup \emptyset$   
 (i)  $(ab)^*a = a(ba)^*$   
 (j)  $(a \cup b)^* b (a \cup b)^* = a^* b (a \cup b)^*$   
 (k)  $[(a \cup b)^* b (a \cup b)^* \cup (a \cup b)^* a (a \cup b)^*] = (a \cup b)^*$   
 (l)  $[(a \cup b)^* b (a \cup b)^* \cup (a \cup b)^* a (a \cup b)^*] = (a \cup b)^+$   
 (m)  $[(a \cup b)^* b a (a \cup b)^* \cup a^*b^*] = (a \cup b)^*$

- (n)  $(L_1L_2L_3)^* = L_1^*L_2^*L_3^*$   
 (o)  $(L_1^* \cup L_3^*) = (L_1^* \cup L_3^*)^*$   
 (p)  $L_1^* L_1 = L_1^+$   
 (q)  $(L_1 \cup L_2)^* = (L_2 \cup L_1)^*$   
 (r)  $L_1^* (L_2 \cup L_3)^+ = (L_1^* L_2^+ \cup L_1^* L_3^+)$   
 (s)  $\emptyset L_1^* = \emptyset$   
 (t)  $\emptyset L_1^* = \{\epsilon\}$   
 (u)  $(L_1 - L_2) = (L_2 - L_1)$   
 (v)  $((L_1 L_2) \cup (L_1 L_3))^* = (L_1 (L_2 \cup L_3))^*$

8. Let  $L = \{w \in \{a, b\}^* : w \text{ contains } bba \text{ as a substring}\}$ . Find a regular expression for  $\{a, b\}^* - L$ .

9. Let  $\Sigma = \{a, b\}$ . For each of the following sets of strings (i.e., languages)  $L$ , first indicate which of the example strings are in the language and which are not. Then, if you can, write a concise description, in English, of the strings that are in the language.

Example strings: (1) aaabbb, (2) abab, (3) abba, (4)  $\epsilon$

(a)  $L = \{w : \text{for some } u \in \Sigma^*, w = u^R u\}$

(b)  $L = \{w : ww = www\}$

(c)  $L = \{w : \text{for some } u \in \Sigma^*, www = uu\}$

10. Write a regular expression for the language consisting of all odd integers *without* leading zeros.

11. Let  $\Sigma = \{a, b\}$ . Let  $L = \{\epsilon, a, b\}$ . Let  $R$  be a relation defined on  $\Sigma^*$  as follows:  $\forall xy, xRy$  iff  $y = xb$ . Let  $R'$  be the reflexive, transitive closure of  $L$  under  $R$ . Let  $L' = \{x : \exists y \in L \text{ such that } yR'x\}$ . Write a regular expression for  $L'$ .

## Solutions

1. (a) Any string of a's and/or b's with zero or more a's followed by a single b.  
 (b) Any string of a's and/or b's with at least one occurrence of ab or ba.

2. (a)  $\emptyset^* = \{\epsilon\}$ , and  $\epsilon \subseteq (a \cup b)^*$ .

$a^* \subseteq (a \cup b)^*$ .

$b^* \subseteq (a \cup b)^*$ . So since the first three terms describe subsets of the last one, unioning them into the last one doesn't add any elements. Thus we can write simply  $(a \cup b)^*$ .

(b) To solve this one, we'll use some identities for regular expressions. We don't have time for an extensive study of such identities, but these are useful ones:

$((a^*b^*)^* (b^*a^*)^*)^* =$

Using  $(A^*B^*)^* = (A \cup B)^*$  (Both simply describe any string that is composed of elements of  $A$  and elements of  $B$  concatenated together in any order)

$((a \cup b)^*(b \cup a)^*)^* =$

Using  $(A \cup B) = (B \cup A)$  (Set union is commutative)

$((a \cup b)^*(a \cup b)^*)^* =$

Using  $A^*A^* = A^*$

$((a \cup b)^*)^* =$

Using  $(A^*)^* = A^*$

$(a \cup b)^*$



(c)  $(a^*b)^* \cup (b^*a)^* = (a \cup b)^*$  (In other words, all strings over  $\{a, b\}$ .) How do we know that?  $(a^*b)^*$  is the union of  $\epsilon$  and all strings that end in  $b$ .  $(b^*a)^*$  is the union of  $\epsilon$  and all strings that end in  $a$ . Clearly any string over  $\{a, b\}$  must either be empty or it must end in  $a$  or  $b$ . So we've got them all.

3. (a) The a's must come in groups of three, but of course there can be arbitrary numbers of b's everywhere. So:

$$(b^*ab^*ab^*a)^*b^*$$

Since the first expression has  $*$  around it, it can occur 0 or more times, to give us any number of a's that is divisible by 3.

(b) Another way to think of this is that there are three optional a's and all the b's you want. That gives us:

$$b^* (a \cup \epsilon) b^* (a \cup \epsilon) b^* (a \cup \epsilon) b^*$$

(c) Another way to think of this is that we need one instance of  $aaa$ . All other instances of  $aa$  must occur with

either  $b$  or end of string on both sides. The  $aaa$  can occur anywhere so we'll plunk it down, then list the options for everything else twice, once on each side of it:

$$(ab \cup aab \cup b)^* \quad aaa \quad (ba \cup baa \cup b)^*$$

4. (a) True. Consider the defining regular expression:  $a^*b^*a^*b^*$ . To get  $baa$ , take no a's, then one  $b$ , then two a's then no b's.

(b) True. We can prove that two sets  $X$  and  $Y$  are equal by showing that any string in  $X$  must also be in  $Y$  and vice versa. First we show that any string in  $b^*a^* \cap a^*b^*$  (which we'll call  $X$ ) must also be in  $a^* \cup b^*$  (which we'll call  $Y$ ). Any string in  $X$  must have two properties: (from  $b^*a^*$ ): all b's come before all a's; and (from  $a^*b^*$ ): all a's come before all b's. The only way to have both of these properties simultaneously is to be composed of only a's or only b's. That's exactly what it takes to be in  $Y$ .

Next we must show that every string in  $Y$  is in  $X$ . Every string in  $Y$  is either of the form  $a^*$  or  $b^*$ . All strings of the form  $a^*$  are in  $X$  since we simply take  $b^*$  to be  $b^0$ , which gives us  $a^* \cap a^* = a^*$ . Similarly for all strings of the form  $b^*$ , where we take  $a^*$  to be  $a^0$ .

(c) False. Remember that to show that any statements is false it is sufficient to find a single counterexample:

$$\epsilon \in a^*b^* \quad \text{and} \quad \epsilon \in c^*d^*. \quad \text{Thus} \quad \epsilon \in a^*b^* \cap c^*d^*, \quad \text{which is therefore not equal to } \emptyset.$$

(d) False. There is no way to generate  $abcd$  from  $(a(cd)^*b)^*$ . Let's call the language generated by  $(a(cd)^*b)^*$   $L$ . Notice that every string in  $L$  has the property that every instance of  $(cd)^*$  is immediately preceded by  $a$ .  $abcd$  does not possess that property.

5. That the language on the right is included in the language on the left is immediately apparent since every string in the right-hand language is a string of a's and b's. To show that any string of a's and b's is contained in the language on the right, we note that any such string begins with zero or more a's. If there are no b's, then the string is contained in  $a^*$ . If there is at least one  $b$ , we strip off any initial a's as a part of  $a^*$  and examine the remainder. If there are no more b's, the remainder is in  $ba^*$ . If there is at least one more  $b$  to the right, then we strip of the initial  $b$  and any following consecutive a's (a string in  $ba^*$ ) and examine the remainder. Repeat the last two steps until the end of the string is reached. Thus, every string of a's and b's is included in the language on the right.

6. (i)  $a, c, e$  (b contains superscript  $n$ ; d contains  $\cap$ )

(ii)  $(a) = (a \cup b)^*$

(c)  $= \emptyset$

(e)  $= b^*$

(iii)  $a, c, d, e$  (b is  $\{a^m b^n : m > n\}$ , which is not regular)

7. (a) F, (b) F, (c) T, (d) F, (e) T, (f) T, (g) T, (h) F, (i) T, (j) T, (k) F, (l) T, (m) T, (n) F, (o) F, (p) T (by def. of +), (q) T, (r) F, (s) T, (t) F, (u) F, (v) T.

8.  $(a \cup ba)^* (\epsilon \cup b \cup bbb^*) = (a \cup ba)^* b^*$

9. (a) (1) no (2) no, (3) yes, (4) yes

L is composed of strings whose second half is the reverse of the first half.

(b) (1) no (2) no (3) no (4) yes

L contains only the empty string.

(c) (1) no (2) yes (3) no (4) yes

L contains strings of even length whose first half is the same as the second half. To see why this is so, notice that  $|uu|$  is necessarily even, since it's  $|u|$  times 2. So we must assure that  $|www|$  is also even. This will only happen if  $|w|$  is even. To discover what  $u$  is for any proposed  $w$ , we must first write out  $www$ . Then we split it in half and call that  $u$ . Suppose that  $w$  can be described as the concatenation of two strings of equal length,  $r$  and  $s$ . (We know we can do this, since we already determined that  $|w|$  is even.) Then  $w$  will be equal to  $rs$  and  $www$  will be  $rsrsrs$ . So  $u$  must equal both  $rsr$  and  $srs$ . There can only be such a  $u$  if  $r$  and  $s$  are the same.

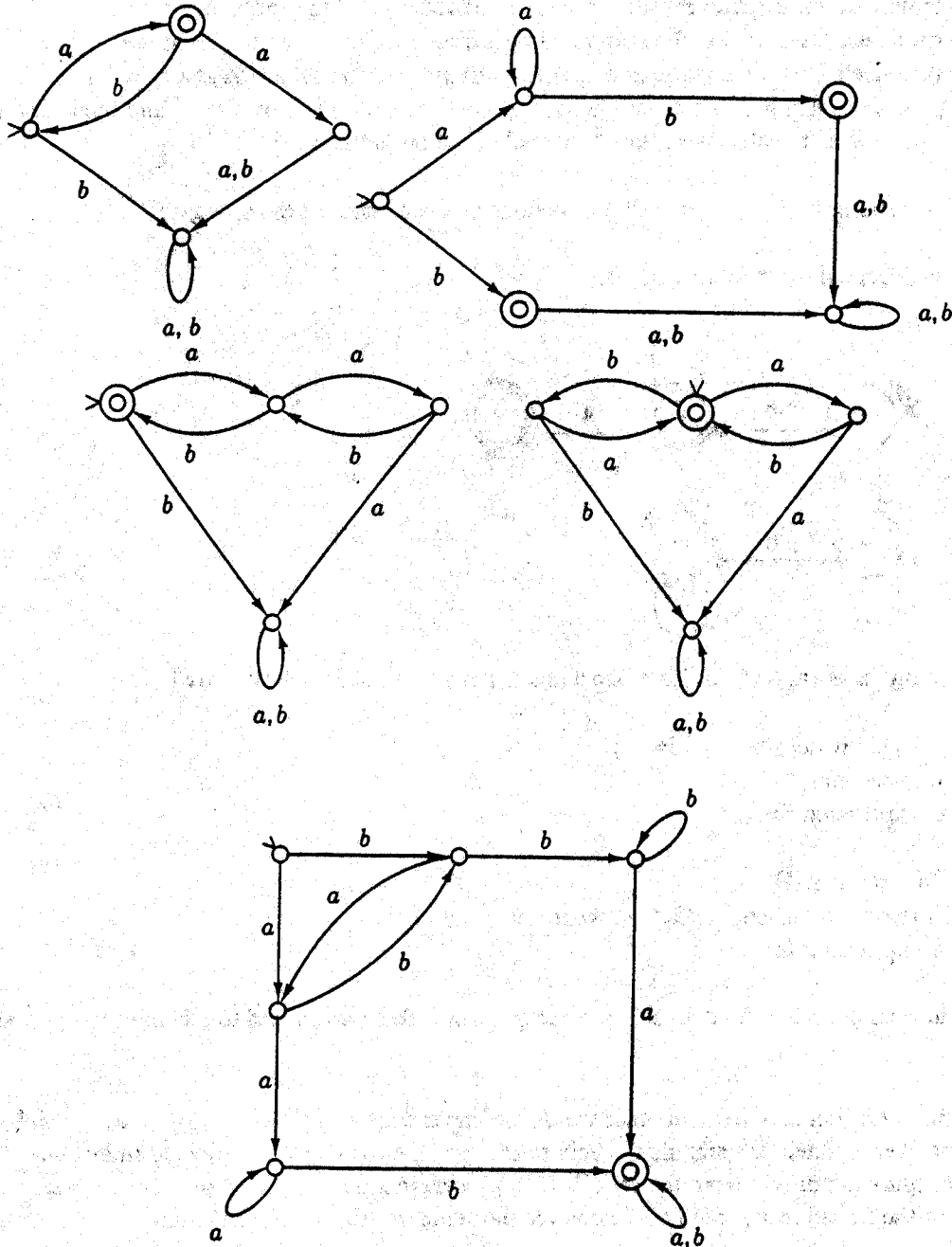
10.  $(\epsilon \cup ((1-9)(0-9)^*)) (1 \cup 3 \cup 5 \cup 7 \cup 9)$ , or, without using  $\epsilon$  or the dash notation,

$$(1 \cup 3 \cup 5 \cup 7 \cup 9) \cup ((1 \cup 2 \cup 3 \cup 4 \cup 5 \cup 6 \cup 7 \cup 8 \cup 9) (0 \cup 1 \cup 2 \cup 3 \cup 4 \cup 5 \cup 6 \cup 7 \cup 8 \cup 9)^* (1 \cup 3 \cup 5 \cup 7 \cup 9))$$

11. Whew. A lot of formalism. The key is to walk through it one step at a time. It's good practice.  $R$  relates pairs of strings that are identical except that the second one has one extra  $b$  concatenated on the end. So it includes, for example,  $\{(a, ab), (ab, abb), (abb, abbb), (b, bb), (bb, bbb), \dots\}$ . Now we have to compute  $R'$ . Consider the element  $a$ . First, we must add  $(a, a)$  to make  $R'$  reflexive. Now we must consider transitivity.  $R$  gives us  $(a, ab)$ . But it also gives us  $(ab, abb)$ , so, by transitivity,  $R'$  must contain  $(a, abb)$ . In fact,  $a$  must be related to all strings in the language  $ab^*$ . Similarly  $\epsilon$  must be related to all strings in  $\epsilon b^*$  or simply  $b^*$ . And  $b$  must be related to all strings in  $bb^*$ . We could also notice many other things, such as the fact that  $ab$  is related to all strings in  $abb^*$ , but we don't need to bother to do that to solve this problem. What we need to do is to figure out what  $L'$  is. It's all strings that are related via  $R'$  to some element of  $L$ . There are three elements of  $L$ ,  $\{\epsilon, a, b\}$ . So  $L' = b^* \cup ab^* \cup bb^*$ . But every string in  $bb^*$  is also in  $b^*$ , so we can simplify to  $b^* \cup ab^*$ .

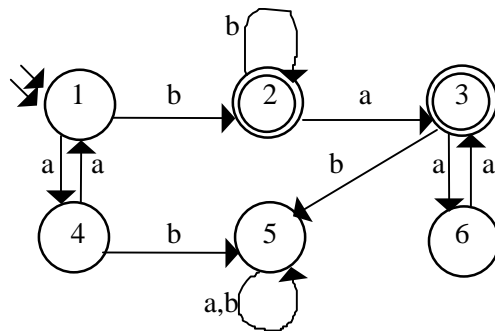
**CS 341 Homework 4**  
**Deterministic Finite Automata**

1. If  $M$  is a deterministic finite automaton. Under exactly what circumstances is  $\epsilon \in L(M)$ ?
2. Describe informally the languages accepted by each of the following deterministic FSMs:



(from Elements of the Theory of Computation, H. R. Lewis and C. H. Papadimitriou, Prentice-Hall, 1998.)

3. Construct a deterministic FSM to accept each of the following languages:
- (a)  $\{w \in \{a, b\}^* : \text{each 'a' in } w \text{ is immediately preceded and followed by a 'b'}\}$
  - (b)  $\{w \in \{a, b\}^* : w \text{ has abab as a substring}\}$
  - (c)  $\{w \in \{a, b\}^* : w \text{ has neither aa nor bb as a substring}\}$
  - (d)  $\{w \in \{a, b\}^* : w \text{ has an odd number of a's and an even number of b's}\}$
  - (e)  $\{w \in \{a, b\}^* : w \text{ has both ab and ba as substrings}\}$
4. Construct a deterministic finite state transducer over  $\{a, b\}$  for each of the following tasks:
- (a) On input  $w$  produce  $a^n$ , where  $n$  is the number of occurrences of the substring  $ab$  in  $w$ .
  - (b) On input  $w$  produce  $a^n$ , where  $n$  is the number of occurrences of the substring  $aba$  in  $w$ .
  - (c) On input  $w$  produce a string of length  $w$  whose  $i^{\text{th}}$  symbol is an  $a$  if  $i = 1$  or if  $i > 1$  and the  $i^{\text{th}}$  and  $(i-1)^{\text{st}}$  symbols of  $w$  are different; otherwise, the  $i^{\text{th}}$  symbol of the output is  $b$ .
5. Construct a dfa accepting  $L = \{w \in \{a, b\}^* : w \text{ contains no occurrence of the string } ab\}$ .
6. What language is accepted by the following fsa?



7. Give a dfa accepting  $\{x \in \{a, b\}^* : \text{at least one a in } x \text{ is not immediately followed by b}\}$ .
8. Let  $L = \{w \in \{a, b\}^* : w \text{ does not end in } ba\}$ .
- (a) Construct a dfa accepting  $L$ .
  - (b) Give a regular expression for  $L$ .
9. Consider  $L = \{a^n b^n : 0 \leq n \leq 4\}$
- (a) Show that  $L$  is regular by giving a dfa that accepts it.
  - (b) Give a regular expression for  $L$ .
10. Construct a deterministic finite state machine to accept strings that correspond to odd integers *without* leading zeros.
11. Imagine a traffic light. Let  $\Sigma = \{a\}$ . In other words, the input consists just of a string of  $a$ 's. Think of each  $a$  as the output from a timer that signals the light to change. Construct a deterministic finite state transducer whose outputs are drawn from the set  $\{Y, G, R\}$  (corresponding to the colors yellow, green, and red). The outputs of the transducer should correspond to the standard traffic light behavior.
12. Recall the finite state machine that we constructed in class to accept \$1.00 in change or bills. Modify the soda machine so that it actually does something (i.e., some soda comes out) by converting our finite state acceptor to a finite state transducer. Let there be two buttons, one for Coke at \$.50 and one for Water at \$.75 (yes, it's strange that water costs more than Coke. The world is a strange place). In any case, there will now be two new symbols in the input alphabet,  $C$  and  $W$ . The machine should behave as follows:

- The machine should keep track of how much money has been inserted. If it ever gets more than \$1.50, it should spit back enough to get it under \$1.00 but keep it above \$.75.
- If the Coke or Water button is pushed and enough money has been inserted, the product and the change should be output.
- If a button is pushed and there is not enough money, the machine should remember the button push and wait until there is enough money, at which point it should output the product and the change.

13. Consider the problem of designing an annoying buzzer that goes off whenever you try to drive your car and you're not wearing a seat belt. (For simplicity, we'll just worry about the driver's possible death wish. If you want to make this harder, you can worry about the other seats as well.) Design a finite state transducer whose inputs are drawn from the alphabet {KI, KR, SO, SU, BF, BU}, representing the following events, respectively: "key just inserted into ignition", "key just removed from ignition", "seat just became occupied", "seat just became unoccupied", "belt has just been fastened", and "belt has just been unfastened". The output alphabet is {ON, OFF}. The buzzer should go on when ON is output and stay off until OFF is output.

14. Is it possible to construct a finite state transducer that can output the following sequence:

1010010001000010000010000001...

If it is possible, design one. If it's not possible, why not?

## Solutions

1.  $\epsilon \in L(M)$  iff the initial state is a final state. Proof:  $M$  will halt in its initial state given  $\epsilon$  as input. So: (IF) If the initial state is a final state, then when  $M$  halts in the initial state, it will be in a final state and will accept  $\epsilon$  as an element of  $L(M)$ . (ONLY IF) If the initial state is not a final state, then when  $M$  halts in the initial state, it will reject its input, namely  $\epsilon$ . So the only way to accept  $\epsilon$  is for the initial state to be a final state.

2.

(a) You must read  $a$  to reach the unique final state. Once there, you may read  $ba$  and still accept. So the language is  $a(ba)^*$ . (Or  $(ab)^*a$ .) This problem is fairly easy to analyze. (Informally, you could describe this as all strings that begin and end with  $a$ , and the symbols alternate  $a$  and  $b$ , or something of this nature; giving the regular expression is much clearer and easier.)

(b) There are two final states that are reachable. This one is quite easy because once you reach the final states you cannot go further. The obvious answer is  $aa^*b \cup b$ . This can be simplified to  $a^*b$ . The machine is distinguishing between whether the number of  $a$ 's is positive or 0, but there is no need to.

(c) This one is trickier. How can we reach the final state here? By going to the middle state with  $a$  and then returning with  $b$ . This can be iterated. But while in the middle state we may iterate  $ab$ . So the answer is  $(a(ab)^*b)^*$ .

(d) This one is similar to (c) but easier. We can reach the final state by reading  $ab$  or  $ba$ , and in either case we may iterate again. So  $(ab \cup ba)^*$  is the solution.

(e) Number the states 1,2,3,4,5,6 going right to left, top to bottom. The following properties characterize each state:

1:  $\epsilon$  has been read.

2:  $xb$  has been read, for some  $x \in (a \cup b)^*$  not ending in  $b$ .

3:  $xbb$  has been read, for some  $x \in (a \cup b)^*$ .

4:  $xa$  has been read, for some  $x \in (a \cup b)^*$  not ending in  $a$ .

5:  $xaa$  has been read, for some  $x \in (a \cup b)^*$ .

6:  $xbbay$  or  $xaaby$  has been read, for some  $x, y \in (a \cup b)^*$ .

Therefore the language is all strings containing  $bba$  or  $aab$  as a substring, i.e.,  $(a \cup b)^*(bba \cup aab)(a \cup b)^*$ .

3.

(a)  $L = \{w \in \{a, b\}^* : \text{each } a \text{ in } w \text{ is immediately preceded and immediately followed by a } b\}$ .

(A regular expression for  $L$  is  $(b^*ba)(b^*ba)^*bb^* \cup b^*$ , or, using  $+$ ,  $(b^+a)^+b^+ \cup b^*$ . Notice the necessary distinction between strings with no  $a$ 's and those with  $a$ 's. Why doesn't the simpler  $b^*(b^+ab^+)^*$  work?)

This will need a machine with a deadstate because as soon as we see an  $a$  not preceded or followed by a  $b$ , the string should be rejected and no matter what comes later, the string is bad. I.e., we will assume the string is ok until a specific occurrence which tells us to reject the string.

Clearly  $\epsilon \in L$  since every  $a$  in  $\epsilon$  has the property. Now for any longer string, the machine only needs to remember what the last symbol was to determine if the string should be rejected.

So we could make states with the properties:

1:  $\epsilon \in L$  has been read.

2:  $xa$  has been read, for some  $x \in L$  not ending in  $a$  (the string so far is ok, but we'd better see a  $b$  next since  $xa \notin L$ .)

3:  $xb \in L$  has been read (the string so far is ok.)

4:  $x$  has been read, such that for no  $y$  is  $xy \in L$ . (we know the string is bad - no matter what comes later.)

You should be able to draw the machine now. Notice that  $s = 1$ ,  $F = \{1, 3\}$ .

(b)  $L = \{w \in \{a, b\}^* : w \text{ has } abab \text{ as a substring}\}$ .

(A regular expression for  $L$  is easy:  $(a \cup b)^*abab(a \cup b)^*$ .)

Again we need to keep track only of the last part of the string, in this case the last 3 symbols. In this one we are looking for an occurrence in the string which *will* make us accept the string (compare to Problem (a).) Once there has been an occurrence of  $abab$ , whatever follows is irrelevant.

Here are the relevant properties of the string as it is read in:

1:  $x$  has been read, for some  $x \in (a \cup b)^*$  such that  $x \notin L$  and  $x$  does not end in  $a$ .

2:  $xa$  has been read, for some  $x \in (a \cup b)^*$  such that  $x \notin L$  and  $x$  does not end in  $ab$ .

3:  $xab$  has been read, for some  $x \in (a \cup b)^*$  such that  $x \notin L$  and  $x$  does not end in  $ab$ .

4:  $xaba$  has been read, for some  $x \in (a \cup b)^*$  such that  $x \notin L$  and  $x$  does not end in  $ab$ .

5:  $xababy$  has been read, for some  $x, y \in (a \cup b)^*$  such that  $x \notin L$  and  $x$  does not end in  $ab$ .

So a 5 state machine can do the trick. The start state is 1, because that's the property  $e$  has ( $e \in (a \cup b)^*$  and  $e$  does not end in  $a$ .) Any string with property 1 which is then followed by  $b$  continues to have property 1, so  $\delta(1, b) = 1$ . Any string with property 1 which is then followed by  $a$  now has property 2, so  $\delta(1, a) = 2$ . And so on. Clearly  $\delta(5, \sigma) = 5$  since once  $abab$  has been seen, that fact cannot be changed -  $abab$  continues to have been seen. Clearly a string has  $abab$  as a substring iff it has property 5, so  $F = \{5\}$ . Now you draw the DFA.

(c)  $L = \{w \in \{a, b\}^* : w \text{ has neither } aa \text{ nor } bb \text{ as a substring}\}$ .

(A regular expression for  $L$  is  $\epsilon \cup a(ba)^*(b \cup \epsilon) \cup b(ab)^*(a \cup \epsilon)$ . This distinguishes between whether the string starts with  $a$  or  $b$  or is empty. Another one is  $(a \cup \epsilon)(ba)^*(b \cup \epsilon)$ , though this is perhaps less obvious.)

Like Problem (a), we should assume the string is ok until we see a bad occurrence ( $aa$  or  $bb$ ). To test this, we clearly only need to keep track of the last symbol read. So the relevant properties are:

- 1:  $e$  has been read (and so  $a$  or  $b$  may follow.)
- 2:  $xa$  has been read, for some  $xa \in L$  (so only  $b$  may follow.)
- 3:  $xb$  has been read, for some  $xb \in L$  (so only  $a$  may follow.)
- 4:  $x$  has been read, for some  $x \notin L$ .

Clearly any string with property 1, 2 or 3 is in  $L$ , so  $F = \{1, 2, 3\}$ . The start state is 1. Now you draw it.

(d)  $L = \{w \in \{a, b\}^* : \#(a, w) \text{ is odd and } \#(b, w) \text{ is even}\}$ .

I use the function  $\#(\sigma, x)$  to mean "the number of occurrences of symbol  $\sigma$  in string  $x$ ." E.g.,  $\#(a, aba) = 2$  and  $\#(b, aaa) = 0$ .

Unlike the previous problems, there is no specific occurrence we are looking for, either to reject or accept the string. Instead, we need to continually monitor it. When the string is all read in, its status will then determine whether it is accepted or rejected.

Clearly what we need to monitor is the parity (even or odd) of the number of  $a$ 's and the number of  $b$ 's. These are independent data, so there are  $2 \times 2 = 4$  possible states or properties:

- (0,0):  $x$  has been read, where  $\#(a, x)$  and  $\#(b, x)$  both even.
- (0,1):  $x$  has been read, where  $\#(a, x)$  even and  $\#(b, x)$  odd.
- (1,0):  $x$  has been read, where  $\#(a, x)$  odd and  $\#(b, x)$  even.
- (1,1):  $x$  has been read, where  $\#(a, x)$  and  $\#(b, x)$  both odd.

Since  $\#(\sigma, \epsilon) = 0$ , and 0 is even, the start state is (0,0). (A fair number of people unnecessarily distinguish between 0 and other even numbers, producing machines with more states than necessary.) The only final state is (1,0).  $\delta$  can be defined by  $\delta((m, n), a) = (m + 1 \bmod 2, n)$  and  $\delta((m, n), b) = (m, n + 1 \bmod 2)$ .

This is a technique easily generalized. Finite automata cannot count to arbitrarily high natural numbers, but they can count modulo a number (so-called "clock arithmetic"). The DFA just given counts the number of  $a$ 's and the number of  $b$ 's modulo 2. (A number  $m$  is even iff  $m$  is congruent to  $0 \bmod 2$ , written  $x \equiv 0 \bmod 2$ , e.g.,  $x = \dots, -2, 0, 2, 4, \dots$ ) You could design a DFA to accept all strings  $x$  with  $\#(a, x) \equiv 7 \bmod 12$  and  $\#(b, x) \equiv 0 \bmod 5$  and  $\#(c, x) \equiv 2 \bmod 3$ , i.e.,  $\#(a, x) = 7, 19, 26, \dots$  and  $\#(b, x)$  is a multiple of 5 and

$\#(c, z) = 2, 5, 8, \dots$ . A minimum state DFA to accept this language uses  $12 \times 5 \times 3 = 180$  states. For notational convenience, I would call the states  $(i, j, k)$ , where  $0 \leq i \leq 12$ ,  $0 \leq j \leq 5$  and  $0 \leq k \leq 3$ . Then the final state would be  $(7, 0, 2)$ . The start state is of course  $(0, 0, 0)$ .

What would you do if you wanted all strings  $x$  with  $\#(a, x) \equiv 2$  or  $3 \pmod 4$ , or  $\#(b, x) \equiv 1 \pmod 3$ ? (Hint: the states are constructed in the same manner; only the final conditions are different.)

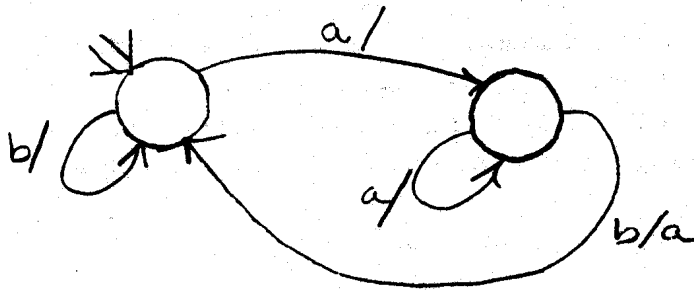
(e)  $L = \{w \in \{a, b\}^* : w \text{ has both } ab \text{ and } ba \text{ as substrings}\}$ .

Here we are looking for not one occurrence but two in the string. There are two subtleties. Either event might occur first, so we must be prepared for the  $ab$  or the  $ba$  to be read first. Also, the definition of  $L$  does not require the two substrings of  $ab$  and  $ba$  to be nonoverlapping: e.g.,  $aba \in L$ .

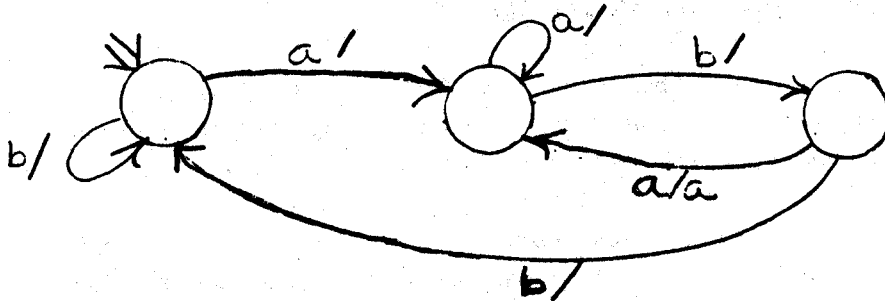
- 1:  $\epsilon$  has been read (so we have seen neither substring.)
- 2:  $a^m$  has been read, for some  $m \geq 1$  (so we have seen neither substring.)
- 3:  $a^m b^n$  has been read, for some  $m, n \geq 1$  (so we have seen  $ab$ .)
- 4:  $b^m$  has been read, for some  $m \geq 1$  (so we have seen neither substring.)
- 5:  $b^m a^n$  has been read, for some  $m, n \geq 1$  (so we have seen  $ba$ .)
- 6:  $a^m b^n a^x$  or  $b^m a^n b^x$  has been read, for some  $m, n \geq 1$  and  $x \in (a \cup b)^*$  (so we have seen  $ab$  and  $ba$ .)

Clearly, 1 is the start state and {6} is the set of final states. You should be able to draw the DFA now.

4. (a)

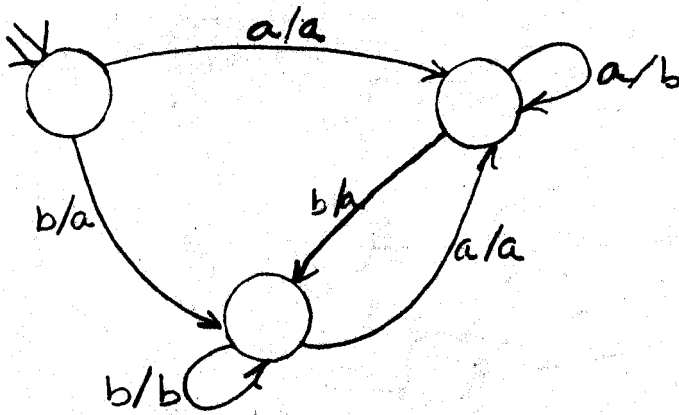


(b)

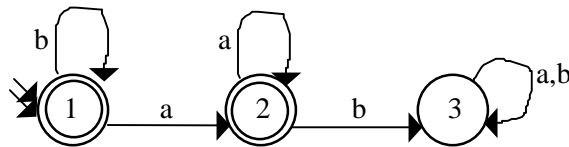




(c)

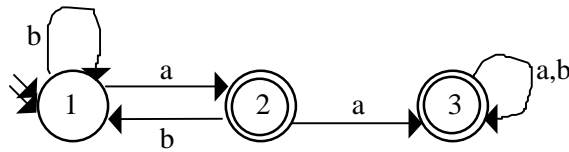


5.



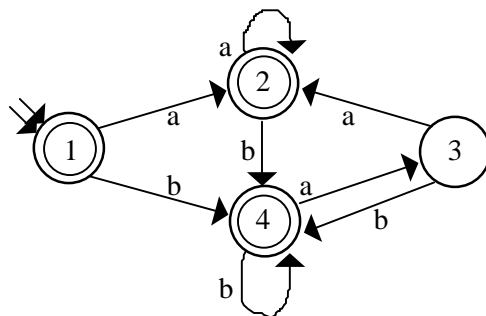
6.  $(aa)^* (bb^* \cup bb^*a(aa)^*) = (aa)^*b^+(\epsilon \cup a(aa)^*)$  = all strings of a's and b's consisting of an even number of a's, followed by at least one b, followed by zero or an odd number of a's.

7.

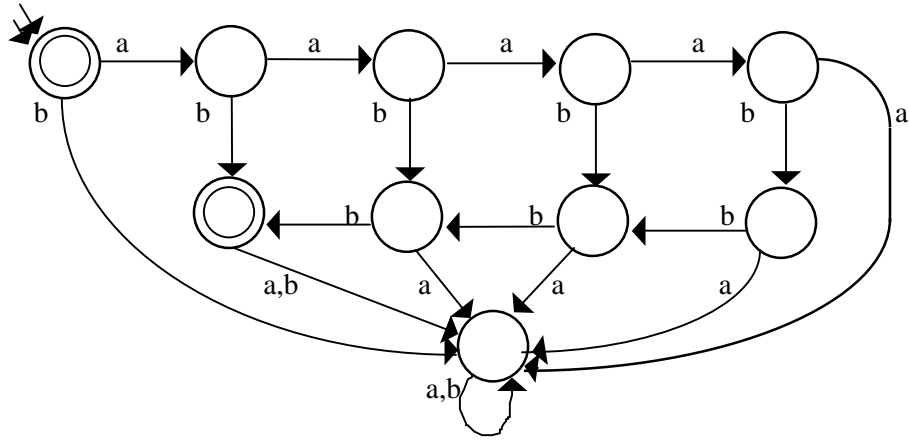


8. (a)

(b)  $\epsilon \cup a \cup (a \cup b)^* (b \cup aa)$



9. (a)



(b)  $(\epsilon \cup ab \cup aabb \cup aaabbb \cup aaaabbbb)$

## CS 341 Homework 5

### Regular Expressions in UNIX

Regular expressions are all over the place in UNIX, including the programs `grep`, `sed`, and `vi`. There's a regular expression pattern matcher built into the programming language `perl`. There's also one built into the `majordomo` maillist program, to be used as a way to filter email messages. So it's easy to see that people have found regular expressions extremely useful. Each of the programs that uses the basic idea offers its own definition of what a regular expression is. Some of them are more powerful than others. The definition in `perl` is shown on the reverse of this page.

1. Write `perl` regular expressions to do the following things. If you have easy access to a `perl` interpreter, you might even want to run them.

- (a) match occurrences of your phone number
- (b) match occurrences of any phone number
- (c) match occurrences of any phone number that occurs more than once in a string
- (d) match occurrences of any email address that occurs more than once in a string
- (e) match the Subject field of any mail message from yourself
- (f) match any email messages where the address of the sender occurs in the body of the message

2. Examine the constructs in the `perl` regular expression definition closely. Compare them to the much more limited definition we are using. Some of them can easily be described in terms of the primitive capabilities we have. In other words, they don't offer additional power, just additional convenience. Some of them, though, are genuinely more powerful, in the sense that they enable you to define languages that aren't regular (i.e., they cannot be recognized with Finite State Machines). Which of the `perl` constructs actually add power to the system? What is it about them that makes them more powerful?

# Regular Expressions in perl

.	Matches any character except newline
[a-z0-9]	Matches any single character of set
[^a-z0-9]	Matches any single character <i>not</i> in set
\d	Matches a digit, same as [0-9]
\D	Matches a non-digit, same as [^0-9]
\w	Matches an alphanumeric (word) character [a-zA-Z0-9_]
\W	Matches a non-word character [^a-zA-Z0-9_]
\s	Matches a whitespace char (space, tab, newline...)
\S	Matches a non-whitespace character
\n	Matches a newline
\r	Matches a return
\t	Matches a tab
\f	Matches a formfeed
\b	Matches a backspace (inside [ ] only)
\0	Matches a null character
\000	Also matches a null character because...
\nnn	Matches an ASCII character of that octal value
\xnn	Matches an ASCII character of that hexadecimal value
\cX	Matches an ASCII control character
\metachar	Matches the character itself (\ , \., \*...)
(abc)	Remembers the match for later backreferences
\1	Matches whatever first of parens matched
\2	Matches whatever second set of parens matched
\3	and so on...
x?	Matches 0 or 1 x's, where x is any of above
x*	Matches 0 or more x's
x+	Matches 1 or more x's
x{m,n}	Matches at least m x's but no more than n
abc	Matches all of a, b, and c in order
fee fie foe	Matches one of fee, fie, or foe
\b	Matches a word boundary (outside [ ] only)
\B	Matches a non-word boundary
^	Anchors match to the beginning of a line or string
\$	Anchors match to the end of a line or string

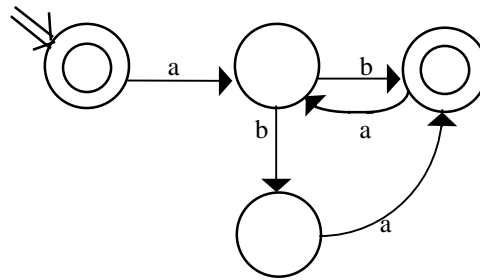
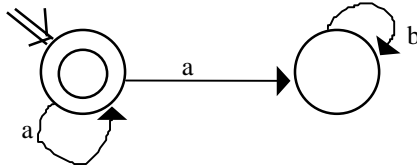
from Programming in Perl, Larry Wall and Randall L. Schwartz, O'Reilly & Associates, 1990.

## CS 341 Homework 6

### Nondeterministic Finite Automata

1. (a) Which of the following strings are accepted by the nondeterministic finite automaton shown on the left below?

- (i) a
- (ii) aa
- (iii) aab
- (iv)  $\epsilon$



(b) Which of the following strings are accepted by the nondeterministic finite automaton on the right above?

- (i)  $\epsilon$
- (ii) ab
- (iii) abab
- (iv) aba
- (v) abaa

2. Write regular expressions for the languages accepted by the nondeterministic finite automata of problem 1.

3. For any FSM  $F$ , let  $|F|$  be the number of states in  $F$ . Let  $R$  be the machine shown on the right in problem 1. Let  $L = \{w \in \{0, 1\}^* : \exists M \text{ such that } M \text{ is an FSM, } L(M) = L(R), |M| \geq |R|, \text{ and } w \text{ is the binary encoding of } |M|\}$ . Write a regular expression for  $L$ .

4. Draw state diagrams for nondeterministic finite automata that accept these languages:

- (a)  $(ab)^*(ba)^* \cup aa^*$
- (b)  $((ab \cup aab)^*a^*)^*$
- (c)  $((a^*b^*a^*)^*b)^*$
- (d)  $(ba \cup b)^* \cup (bb \cup a)^*$

5. Some authors define a nondeterministic finite automaton to be a quintuple  $(K, \Sigma, \Delta, S, F)$ , where  $K, \Sigma, \Delta$ , and  $F$  are as we have defined them and  $S$  is a finite set of initial states, in the same way that  $F$  is a finite set of final states. The automaton may nondeterministically begin operating in any of these initial states. Explain why this definition is not more general than ours in any significant way.

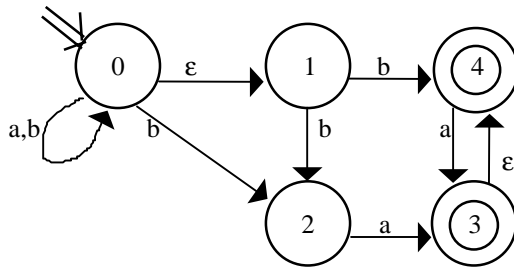
6. (a) Find a simple nondeterministic finite automaton accepting  $((a \cup b)^*aabab)$ .

(b) Convert the nondeterministic finite automaton of Part (a) into a deterministic finite automaton by the method described in class and in the notes.

(c) Try to understand how the machine constructed in Part (b) operates. Can you find an equivalent deterministic machine with fewer states?

7. Construct a NDFSA that accepts the language  $(ba \cup ((a \cup bb) a^*b))$ .

8. Construct a deterministic finite automaton equivalent to the following nondeterministic automaton:



9.  $L = \{ w \in \{a, b\}^* : \text{every } a \text{ is followed by at least one } b \}$
- Write a regular expression that describes  $L$ .
  - Write a regular grammar that describes  $L$ .
  - Construct an FSM that accepts precisely  $L$ .

10. Consider the following regular grammar, which defines a language  $L$ :

- $S \rightarrow bF$
- $S \rightarrow aS$
- $F \rightarrow \epsilon$
- $F \rightarrow bF$
- $F \rightarrow aF$

- Construct an FSM that accepts precisely  $L$ .
- Write a regular expression that describes  $L$ .
- Describe  $L$  in English.

### Solutions

1. (a) [i.] yes [ii.] yes [iii.] no [iv.] yes  
 (b) [i.] yes [ii.] yes [iii.] yes [iv.] yes [v.] no

2. (a)  $a^*$  Note that the second state could be eliminated, since there's no path from it to a final state.  
 (b)  $(ab \cup aba)^*$  Notice that we could eliminate the start state and make the remaining final state the start state and we'd still get the same result.

3. To determine  $L$ , we need first to consider the set of machines that accept the same language as  $R$ . It turns out that we don't actually need to know what all such machines look like because we can immediately see that there's at least one with four states ( $R$ ), one with 5, one with 6, and so forth, and all that we need to establish  $L$  is the sizes of the machines, not their structures.. From  $R$ , we can construct an infinite number of equivalent machines by adding any number of redundant states. For example, we could add a new, nonfinal state 1 that is reachable from the start state via an  $\epsilon$  transition. Since 1 is not final and it doesn't go anywhere, it cannot lead to an accepting path, so adding it to  $R$  has no effect on  $R$ 's behavior. Now we have an equivalent machine with 5 states. We can do it again to yield 6, and so forth. Thus the set of numbers we need to represent is simply  $4 \leq n$ . Now all we have to do is to describe the binary encodings of these numbers. If we want to disallow leading zeros, we get  $1(0 \cup 1)(0 \cup 1)(0 \cup 1)^*$ . There must be at least three digits of which the first must be 1.

4. (a) The easiest way to do this is to make a 2 state FSA for  $aa^*$  and a 4 state one for  $(ab)^*(ba)^*$ , then make a seventh state, the start state, that nondeterministically guesses which class an input string will fall into.

(b) First we simplify.  $((ab \cup aab)^*a^*)^*$

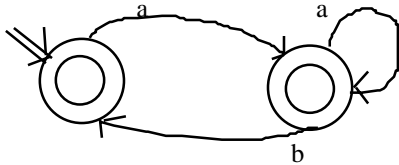
$$/ (L_1^*L_2^*)^* = (L_1 \cup L_2)^* /$$

$$((ab \cup aab) \cup a)^*$$

/ union is associative /

$$(ab \cup aab \cup a)^*, \text{ which can be rewritten as}$$

$(ab \cup a)^*$ . This is so because aab can be formed by one application a, followed by one of ab. So it is redundant inside a Kleene star. Now we can write a two state machine:



If you put the loop on a on the start state, either in place of where we have it, or in addition to it, it's also right.

(c) First we simplify:  $((a^*b^*a^*)^*b)^*$

$$/ (L_1^*L_2^*L_3^*)^* = (L_1 \cup L_2 \cup L_3)^* /$$

$$((a \cup b \cup a)^*b)^*$$

/ union is idempotent /

$$((a \cup b)^*b)^*$$

There is a simple 2 state NDFSM accepting this, which is the empty string and all strings ending with b.

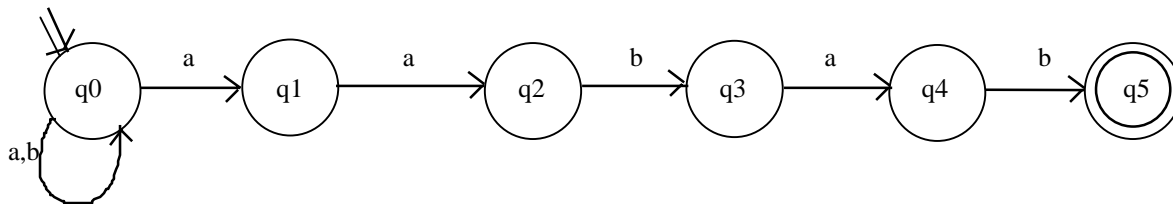
(d) This is the set of strings where either: (1) every a is preceded by a b,  
or (2) all b's occur in pairs.

So we can make a five state nondeterministic machine by making separate machines (each with two states) for the two languages and then introducing  $\epsilon$  transitions from the start state to both of them.

5. To explain that any construct A is not more general or powerful than some other construct B, it suffices to show that any instance of A can be simulated by a corresponding instance of B. So in this case, we have to show how to take a multiple start state NDFSA, A, and convert it to a NDFSA, B, with a single start state. We do this by initially making B equal to A. Then add to B a new state we'll call  $S_0$ . Make it the only start state in B. Now add  $\epsilon$  transitions from  $S_0$  to each of the states that was a start state in A. So B has a single start state (thus it satisfies our original definition of a NDFSA), but it simulates the behavior of A since the first thing it does is to move, nondeterministically, to all of A's start states and then it exactly mimics the behavior of A.

6. If you take the state machine as it is given, add a new start state and make  $\epsilon$  transitions from it to the given start states, you have an equivalent machine in the form that we've been using.

7. (a)



(b) (1) Compute the  $E(q)$ s. Since there are no  $\epsilon$  transitions,  $E(q)$ , for all states  $q$  is just  $\{q\}$ .

(2)  $S' = \{q_0\}$

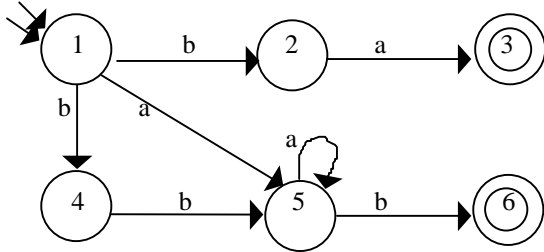
(3)  $\delta' = \{$

- $(\{q_0\}, a, \{q_0, q_1\}),$
- $(\{q_0\}, b, \{q_0\}),$
- $(\{q_0, q_1\}, a, \{q_0, q_1, q_2\}),$
- $(\{q_0, q_1\}, b, \{q_0\}),$
- $(\{q_0, q_1, q_2\}, a, \{q_0, q_1, q_2\}),$
- $(\{q_0, q_1, q_2\}, b, \{q_0, q_3\}),$
- $(\{q_0, q_3\}, a, \{q_0, q_1, q_4\}),$
- $(\{q_0, q_3\}, b, \{q_0\}),$
- $(\{q_0, q_1, q_4\}, a, \{q_0, q_1, q_2\}),$
- $(\{q_0, q_1, q_4\}, b, \{q_0, q_5\}),$
- $(\{q_0, q_5\}, a, \{q_0, q_1\}),$
- $(\{q_0, q_5\}, b, \{q_0\})$  }

- (4)  $K' = \{\{q_0\}, \{q_0, q_1\}, \{q_0, q_1, q_2\}, \{q_0, q_1, q_2, q_3\}, \{q_0, q_1, q_2, q_3, q_4\}, \{q_0, q_1, q_2, q_3, q_4, q_5\}\}$
- (5)  $F' = \{\{q_0, q_5\}\}$

(c) There isn't a simpler machine since we need a minimum of six states in order to keep track of how many characters (between 0 and 5) of the required trailing string we have seen so far.

8. We can build the following machine really easily. We make the path from 1 to 2 to 3 for the ba option. The rest is for the second choice. We get a nondeterministic machine, as we generally do when we use the simple approach.

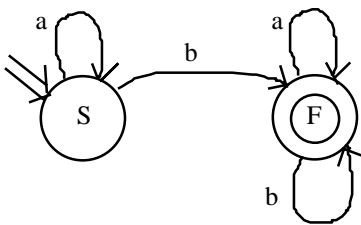


In this case, we could simplify our machine if we wanted to and get rid of state 4 by adding a transition on b from 2 to 5.

- 9. (1)  $E(q_0) = \{q_0, q_1\}, E(q_1) = \{q_1\}, E(q_2) = \{q_2\}, E(q_3) = \{q_3, q_4\}, E(q_4) = \{q_4\}$
- (2)  $s' = \{q_0, q_1\}$
- (3)  $\delta' = \{ (\{q_0, q_1\}, a, \{q_0, q_1\}), (\{q_0, q_1\}, b, \{q_0, q_1, q_2, q_4\}), (\{q_0, q_1, q_2, q_4\}, a, \{q_0, q_1, q_3, q_4\}), (\{q_0, q_1, q_2, q_4\}, b, \{q_0, q_1, q_2, q_4\}), (\{q_0, q_1, q_3, q_4\}, a, \{q_0, q_1, q_3, q_4\}), (\{q_0, q_1, q_3, q_4\}, b, \{q_0, q_1, q_2, q_4\}) \}$
- (4)  $K' = \{ \{q_0, q_1\}, \{q_0, q_1, q_3, q_4\}, \{q_0, q_1, q_2, q_4\} \}$
- (5)  $F' = \{ \{q_0, q_1, q_3, q_4\}, \{q_0, q_1, q_2, q_4\} \}$

This machine corresponds to the regular expression  $a^*b(a \cup b)^*$

10. (a)



(b)  $(a \cup b)^*ba^*$  OR  $a^*b(a \cup b)^*$

(c)  $L = \{ w \in \{a, b\}^* : \text{there is at least one } b \}$



## CS 341 Homework 7 Review of Equivalence Relations

1. Assume a finite domain that includes just the specific cities mentioned here. Let  $R$  = the reflexive, symmetric, transitive closure of:

(Austin, Dallas), (Dallas, Houston), (Dallas, Amarillo), (Austin, San Marcos),  
(Philadelphia, Pittsburgh), (Philadelphia, Paoli), (Paoli, Scranton),  
(San Francisco, Los Angeles), (Los Angeles, Long Beach), (Long Beach, Carmel)

(a) Draw  $R$  as a graph.

(b) List the elements of the partition defined by  $R$  on its domain.

2. Let  $R$  be a relation on the set of positive integers. Define  $R$  as follows:

$\{(a, b) : (a \bmod 2) = (b \bmod 2)\}$  In other words,  $R(a, b)$  iff  $a$  and  $b$  have the same remainder when divided by 2.

(a) Consider the following example integers: 1, 2, 3, 4, 5, 6. Draw the subset of  $R$  involving just these values as a graph.

(b) How many elements are there in the partition that  $R$  defines on the positive integers?

(c) List the elements of that partition and show some example elements.

3. Consider the language  $L$ , over the alphabet  $\Sigma = \{a, b\}$ , defined by the regular expression

$a^*(b \cup \epsilon) a^*$

Let  $R$  be a relation on  $\Sigma^*$ , defined as follows:

$R(x, y)$  iff both  $x$  and  $y$  are in  $L$  or neither  $x$  nor  $y$  is in  $L$ . In other words,  $R(x, y)$  if  $x$  and  $y$  have identical status with respect to  $L$ .

(a) Consider the following example elements of  $\Sigma^*$ :  $\epsilon$ ,  $b$ ,  $aa$ ,  $bb$ ,  $aabaaa$ ,  $bab$ ,  $bbaabb$ . Draw the subset of  $R$  involving just these values as a graph.

(b) How many elements are there in the partition that  $R$  defines on  $\Sigma^*$ ?

(c) List the elements of that partition and show some example elements.

### Solutions

1. (b) [cities in Texas], [cities in Pennsylvania], [cities in California]

2. (b) Two

(c) [even integers] Examples: 2, 4, 6, 106

[odd integers] Examples: 1, 3, 5, 17, 11679

3. (a) (Hint:  $L$  is the language of strings with no more than one  $b$ .)

(b) Two

(c) [strings in  $L$ ] Examples:  $\epsilon$ ,  $aa$ ,  $b$ ,  $aabaaa$

[strings not in  $L$ ] Examples:  $bb$ ,  $bbaabb$ ,  $bab$

## CS 341 Homework 8

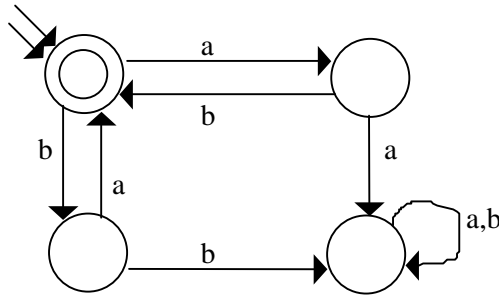
### Finite Automata, Regular Expressions, and Regular Grammars

1. We showed that the set of finite state machines is closed under complement. To do that, we presented a technique for converting a *deterministic* machine  $M$  into a machine  $M'$  such that  $L(M')$  is the complement of  $L(M)$ . Why did we insist that  $M$  be deterministic? What happens if we interchange the final and nonfinal states of a nondeterministic finite automaton?

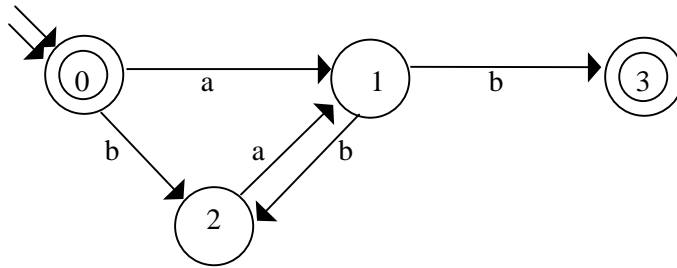
2. Give a direct construction for the closure under intersection of the languages accepted by finite automata. (Hint: Consider an automaton whose set of states is the Cartesian product of the sets of states of the two original automata.) Which of the two constructions, the one given in the text or the one suggested in this problem, is more efficient when the two languages are given in terms of nondeterministic finite automata?

3. Using either of the construction techniques that we discussed, construct a finite automaton that accepts the language defined by the regular expression:  $a^*(ab \cup ba \cup \epsilon)b^*$ .

4. Write a regular expression for the language recognized by the following FSM:



5. Consider the following FSM  $M$ :



- (a) Write a regular expression for the language accepted by  $M$ .
- (b) Give a deterministic FSM that accepts the complement of the language accepted by  $M$ .

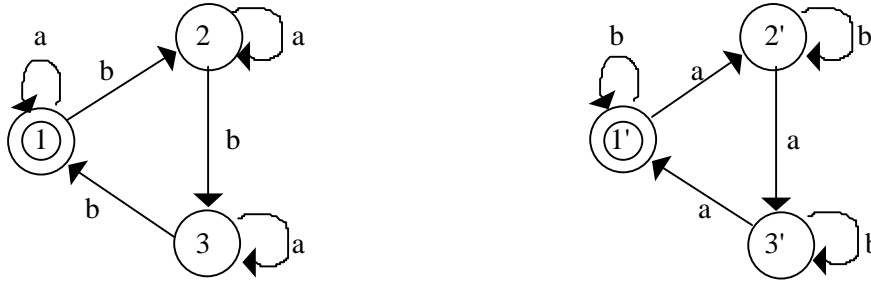
6. Construct a deterministic FSM to accept each of the following languages:

- (a)  $(aba \cup aabaa)^*$
- (b)  $(ab)^*(aab)^*$

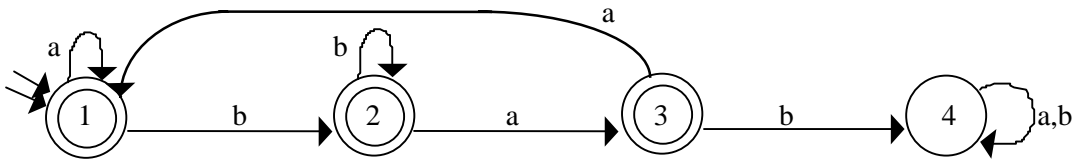
7. Consider the language  $L = \{w \in (a, b)^* : w \text{ has an odd number of } a\text{'s}\}$

- (a) Write a regular grammar for  $L$ .
- (b) Use that grammar to derive a (possibly nondeterministic) FSA to accept  $L$ .

8. Construct a deterministic FSM to accept the intersection of the languages accepted by the following FSMs:



9. Consider the following FSM M:



- (a) Give a regular expression for  $L(M)$ .
- (b) Describe  $L(M)$  in English.

**Solutions**

1. We define acceptance for a NDFSA corresponding to the language L as there existing at least one path that gets us to a final state. There can be many other paths that don't, but we ignore them. So, for example, we might accept a string S that gets us to three different states, one of which accepts (which is why we accept the string) and two of which don't (but we don't care). If we simply flip accepting and nonaccepting states to get a machine that represents the complement of L, then we still have to follow all possible paths, so that same string S will get us to one nonaccepting state (the old accepting state), and two accepting states (the two states that previously were nonaccepting but we ignored). Unfortunately, we could ignore the superfluous nonaccepting paths in the machine for L, but now that those same paths have gotten us to accepting states, we can't ignore them, and we'll have to accept S. In other words, we'll accept S as being in the complement of L, even though we also accepted it as being in L. The key is that in a deterministic FSA, a rejecting path actually means reject. Thus it makes sense to flip it and accept if we want the complement of L. In a NDFSA, a rejecting path doesn't actually mean reject. So it doesn't make sense to flip it to an accepting state to accept the complement of L.

2.

Given two DFA's  $M_1 = (K_1, \Sigma, \delta_1, s_1, F_1)$  and  $M_2 = (K_2, \Sigma, \delta_2, s_2, F_2)$ , we wish to construct a new machine  $M = (K, \Sigma, \delta, s, F)$  such that  $L(M) = L(M_1) \cap L(M_2)$ . (Notice that of course the alphabets of the 3 DFA's will be equal.)

Since the regular languages are closed under union and complementation, and since  $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ , closure under intersection is already proved. This direct construction will avoid using the earlier constructions and illustrates a different proof technique.

The hint is to let  $K = K_1 \times K_2$ . Thus each state of M is really a pair  $(q_1, q_2)$  of states from  $M_1$  and  $M_2$ . The intuition will be that M simultaneously simulates  $M_1$  and  $M_2$  on a given input string. M will keep track of what states  $M_1$  and  $M_2$  would be in if they were reading the string. These are two independent pieces of data; hence the use of a pair for M's state.

Initially,  $M_1$  and  $M_2$  start in their start states,  $s_1$  and  $s_2$ . Therefore we should let  $s = (s_1, s_2)$ .

Now suppose that  $M_1$  is in some state  $q_1 \in K_1$  and reads symbol  $\sigma$ . What state does  $M_1$  enter?  $\delta_1(q_1, \sigma)$ . Similarly for  $M_2$ . So we would like  $M$ , when in state  $(q_1, q_2)$  and reading  $\sigma$ , to enter state  $(\delta_1(q_1, \sigma), \delta_2(q_2, \sigma))$ ; otherwise  $M$  would not be correctly keeping track of what  $M_1$  and  $M_2$  would do. So we define, for all  $(q_1, q_2) \in K$  and all  $\sigma \in \Sigma$ ,

$$\delta((q_1, q_2), \sigma) = (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)).$$

Notice that  $\delta : K \times \Sigma \rightarrow K$ , so everything is consistent and correct. Since  $K = K_1 \times K_2$ , this means  $\delta$  is actually a function taking a pair of states (from  $M_1$  and  $M_2$ ) and a symbol from  $\Sigma$ .

We've now got the transitions defined, and  $M$  correctly simulates  $M_1$  and  $M_2$ . I.e.,

$$\delta(s, x) = (q_1, q_2)$$

iff

$$\delta_1(s_1, x) = q_1 \text{ and } \delta_2(s_2, x) = q_2.^1$$

So we only need to define  $F$ . When should  $M$  accept  $x$ ? Exactly when both  $M_1$  and  $M_2$  do, since  $x \in L(M_1) \cap L(M_2)$  iff  $x \in L(M_1)$  and  $x \in L(M_2)$ . Therefore  $F$  should consist of all those states  $(q_1, q_2) \in K$  such that  $q_1 \in F_1$  and  $q_2 \in F_2$ . This can be written as

$$F = \{(q_1, q_2) : q_1 \in F_1 \text{ and } q_2 \in F_2\},$$

or more succinctly as  $F = F_1 \times F_2$ .

Thus the complete answer is

$$M = (K_1 \times K_2, \Sigma, \delta, (s_1, s_2), F_1 \times F_2)$$

where

$$\delta((q_1, q_2), \sigma) = (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)).$$

Notice that this assumes  $M_1$  and  $M_2$  are deterministic. What if  $M_1$  and  $M_2$  are not deterministic? We can assume that they are deterministic without loss of generality, because if they were not, the subset construction can be applied to them to produce equivalent DFA's. However, this construction can be modified to work directly on NFA's if desired. Unfortunately, it gets rather messy because of the following problem:

We are given two NFA's  $M_1 = (K_1, \Sigma, \Delta_1, s_1, F_1)$  and  $M_2 = (K_2, \Sigma, \Delta_2, s_2, F_2)$ , and we wish to construct a new machine  $M = (K, \Sigma, \Delta, s, F)$  such that  $L(M) = L(M_1) \cap L(M_2)$ .

---

<sup>1</sup>Technically,  $\delta$  is a function of symbols not strings; however we can easily extend it to strings by the recursive generalization:

$$\begin{aligned} \delta(q, \epsilon) &= q \\ \delta(q, \sigma x) &= \delta(\delta(q, \sigma), x) \end{aligned}$$

I.e., if it is determined what  $\delta$  does with a single symbol, then it is determined what  $\delta$  does with a string simply by tracing through symbol by symbol.

If we do the obvious thing and define

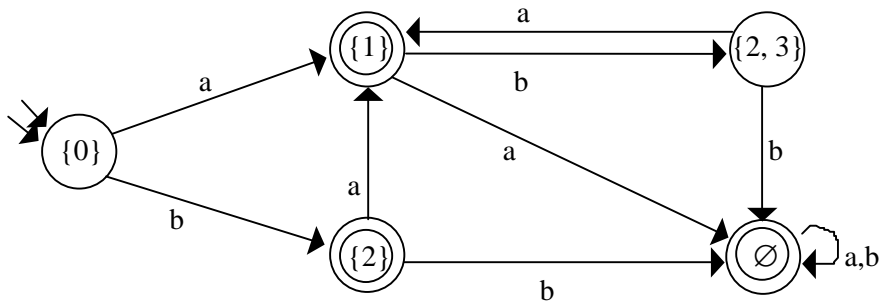
$$\Delta = \{((q_1, q_2), x, (q'_1, q'_2)) : (q_1, x, q'_1) \in \Delta_1 \text{ and } (q_2, x, q'_2) \in \Delta_2\},$$

i.e., we make a transition  $(q_1, q_2) \xrightarrow{x} (q'_1, q'_2)$  in  $M$  exactly when there are transitions  $q_1 \xrightarrow{x} q'_1$  in  $M_1$  and  $q_2 \xrightarrow{x} q'_2$  in  $M_2$ , then there is trouble. The trouble is that the transitions in an NFA need not read exactly 1 symbol, so  $M$  defined this way will be unable to simulate many of moves of  $M_1$  and  $M_2$ . E.g., if  $M_1$  has the transition  $(s_1, aa, q_1)$  and  $M_2$  has  $(s_2, a, q_2)$ , you can see that  $M$  will have difficulty keeping in synch. So  $\Delta$  will have to be defined much more cleverly (and complexly). So it's much easier to just assume  $M_1$  and  $M_2$  are deterministic.

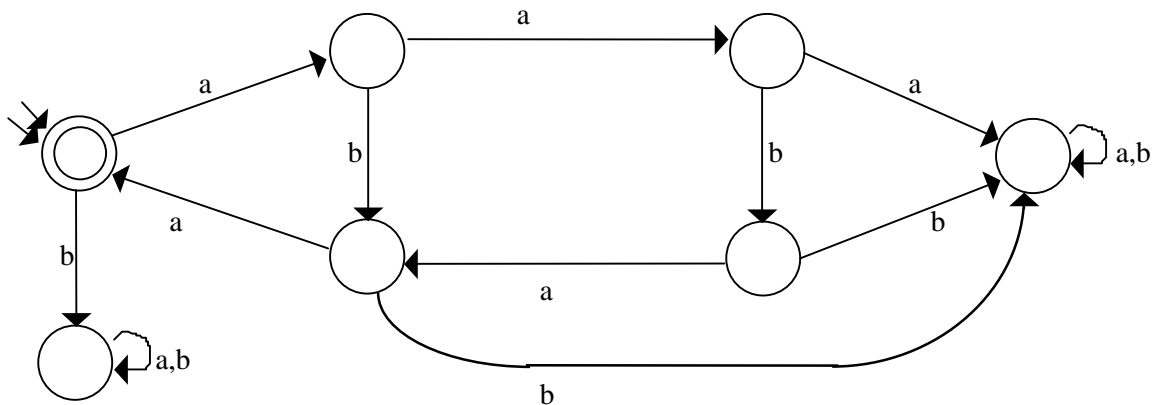
4. Without using the algorithm for finding a regular expression from an FSM, we can note in this case that the lower right state is a dead state, i.e., an absorbing, non-accepting state. We can leave and return to the initial state, the only accepting state, by reading  $ab$  along the upper path or by reading  $ba$  along the lower path. These can be read any number of times, in any order, so the regular expression is  $(ab \cup ba)^*$ . Note that  $\epsilon$  is included, as it should be.

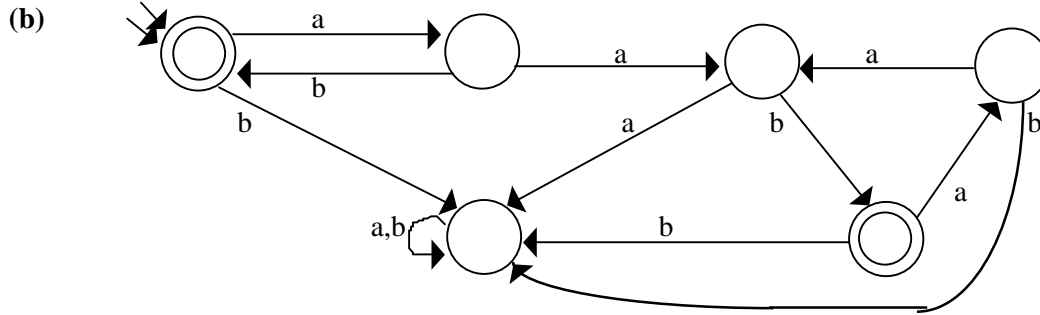
5. (a)  $\epsilon \cup ((a \cup ba)(ba)^*b)$

(b)



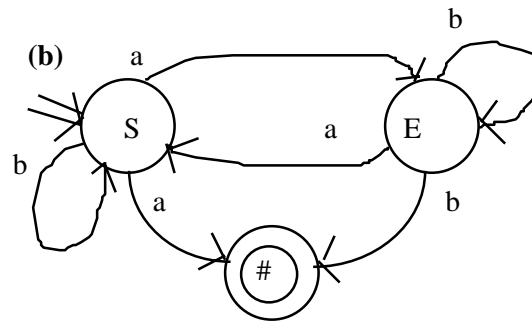
6. (a)



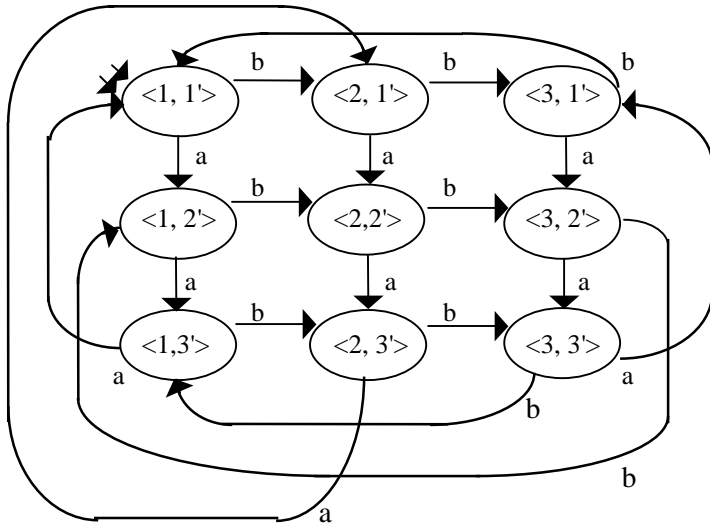


7. (a) Nonterminal S is the starting symbol. We'll use it to generate an odd number of a's. We'll also use the nonterminal E, and it will always generate an even number of a's. So, whenever we generate an a, we must either stop then, or we must generate the nonterminal E to reflect the fact that if we generate any more a's, we must generate an even number of them.

- $S \rightarrow a$
- $S \rightarrow aE$
- $S \rightarrow bS$
- $E \rightarrow b$
- $E \rightarrow bE$
- $E \rightarrow aS$



8.



9. (a)  $(a \cup bb^*aa)^*(\epsilon \cup bb^*(a \cup \epsilon))$

(b) All strings in  $\{a, b\}^*$  that contain no occurrence of bab.

## CS 341 Homework 9

### Languages That Are and Are Not Regular

1. Show that the following are not regular.

(a)  $L = \{ww^R : w \in \{a, b\}^*\}$

(b)  $L = \{ww : w \in \{a, b\}^*\}$

(c)  $L = \{ww' : w \in \{a, b\}^*\}$ , where  $w'$  stands for  $w$  with each occurrence of  $a$  replaced by  $b$ , and vice versa.

2. Show that each of the following is or is not a regular language. The decimal notation for a number is the number written in the usual way, as a string over the alphabet  $\{-, 0, 1, \dots, 9\}$ . For example, the decimal notation for 13 is a string of length 2. In unary notation, only the symbol 1 is used; thus 5 would be represented as 11111 in unary notation.

(a)  $L = \{w : w \text{ is the unary notation for a natural number that is a multiple of } 7\}$

(b)  $L = \{w : w \text{ is the decimal notation for a natural number that is a multiple of } 7\}$

(c)  $L = \{w : w \text{ is the unary notation for a natural number } n \text{ such that there exists a pair } p \text{ and } q \text{ of twin primes, both } > n.\}$  Two numbers  $p$  and  $q$  are a pair of twin primes iff  $q = p + 2$  and both  $p$  and  $q$  are prime. For example,  $(3, 5)$  is a pair of twin primes.

(d)  $L = \{w : w \text{ is, for some } n \geq 1, \text{ the unary notation for } 10^n\}$

(e)  $L = \{w : w \text{ is, for some } n \geq 1, \text{ the decimal notation for } 10^n\}$

(f)  $L = \{w \text{ is of the form } x\#y, \text{ where } x, y \in \{1\}^+ \text{ and } y = x+1 \text{ when } x \text{ and } y \text{ are interpreted as unary numbers}\}$   
(For example,  $11\#111$  and  $1111\#11111 \in L$ , while  $11\#11$ ,  $1\#111$ , and  $1111 \notin L$ .)

(g)  $L = \{a^n b^j : |n - j| = 2\}$

(h)  $L = \{uww^Rv : u, v, w \in \{a, b\}^+\}$

(i)  $L = \{w \in \{a, b\}^* : \text{for each prefix } x \text{ of } w, \#a(x) \geq \#b(x)\}$

3. Are the following statements true or false? Explain your answer in each case. (In each case, a fixed alphabet  $\Sigma$  is assumed.)

(a) Every subset of a regular language is regular.

(b) Let  $L' = L_1 \cap L_2$ . If  $L'$  is regular and  $L_2$  is regular,  $L_1$  must be regular.

(c) If  $L$  is regular, then so is  $L' = \{xy : x \in L \text{ and } y \notin L\}$ .

(d)  $\{w : w = w^R\}$  is regular.

(e) If  $L$  is a regular language, then so is  $L' = \{w : w \in L \text{ and } w^R \in L\}$ .

(f) If  $C$  is any set of regular languages,  $\cup C$  (the union of all the elements of  $C$ ) is a regular language.

(g)  $L = \{xyx^R : x, y \in \Sigma^*\}$  is regular.

(h) If  $L' = L_1 \cup L_2$  is a regular language and  $L_1$  is a regular language, then  $L_2$  is a regular language.

(i) Every regular language has a regular proper subset.

(j) If  $L_1$  and  $L_2$  are nonregular languages, then  $L_1 \cup L_2$  is also not regular.

4. Show that the language  $L = \{a^n b^m : n \neq m\}$  is not regular.

5. Prove or disprove the following statement:

If  $L_1$  and  $L_2$  are not regular languages, then  $L_1 \cup L_2$  is not regular.

6. Show that the language  $L = \{x \in \{a, b\}^* : x = a^n b^m b^{\max(m,n)}\}$  is not regular.

7. Show that the language  $L = \{x \in \{a, b\}^* : x \text{ contains exactly two more } b\text{'s than } a\text{'s}\}$  is not regular.

8. Show that the language  $L = \{x \in \{a, b\}^* : x \text{ contains twice as many } a\text{'s as } b\text{'s}\}$  is not regular.

9. Let  $L = \{w : \#a(w) = \#b(w)\}$ . ( $\#a(w)$  = the number of a's in  $w$ .)

(a) Is  $L$  regular?

(b) Is  $L^*$  regular?

### Solutions

1. (a)  $L = \{ww^R : w \in \{a, b\}^*\}$ .  $L$  is the set of all strings whose first half is equal to the reverse of the second half. All strings in  $L$  must have even length. If  $L$  is regular, then the pumping lemma tells us that  $\exists N \geq 1$ , such that  $\forall$  strings  $w \in L$ , where  $|w| \geq N$ ,  $\exists x, y, z$ , such that  $w = xyz$ ,  $|xy| \leq N$ ,  $y \neq \epsilon$ , and  $\forall q \geq 0$ ,  $xy^qz$  is in  $L$ . We must pick a string  $w \in L$  and show that it does not meet these requirements.

First, don't get confused by the fact that we must pick a string  $w$ , yet we are looking for strings of the form  $ww^R$ . These are two independent uses of the variable name  $w$ . It just happens that the problem statement uses the same variable name that the pumping lemma does. If it helps, restate the problem as  $L = \{ss^R : s \in \{a, b\}^*\}$ .

We need to choose a "long"  $w$ , i.e., one whose length is greater than  $N$ . But it may be easier if we choose one that is even longer than that. Remember that the fact that  $|xy| \leq N$  guarantees that  $y$  (the pumpable region) must occur within the first  $N$  characters of  $w$ . If we don't want to have to consider a lot of different possibilities for what  $y$  could be, it will help to choose a  $w$  with a long first region. Let's let  $w = a^N b b a^N$ . We know that  $y$  must consist of one or more a's in the region before the b's. Clearly if we pump in any extra a's, we will no longer have a string in  $L$ . Thus we know that  $L$  is not regular.

Notice that we could have skipped the b's altogether and chosen  $w = a^N a^N$ . Again, we'd know that  $y$  must be a string of one or more a's. Unfortunately, if  $y$  is of even length (and it could be: remember we don't get to pick  $y$ ), then we can pump in all the copies of  $y$  we want and still have a string in  $L$ . Sure, the boundary between the first half and the second half will move, that that doesn't matter. It is usually good to choose a string with a long, uniform first region followed by a definitive boundary between it and succeeding regions so that when you pump, it's clearly the first region that has changed.

(b)  $L = \{ww : w \in \{a, b\}^*\}$ . We'll use the pumping lemma. Again, don't get confused by the use of the variable  $w$  both to define  $L$  and as the name for the string we will choose to pump on. As is always the case, the only real work we have to do is to choose an appropriate string  $w$ . We need one that is long enough (i.e.,  $|w| \geq N$ ). And we need one with firm boundaries between regions. So let's choose  $w = a^N b a^N b$ . Since  $|xy| \leq N$ , we know that  $y$  must occur in the first  $a$  region. Clearly if we pump in any additional a's, the two halves of  $w$  will no longer be equal. Q. E. D. By the way, we could have chosen other strings for  $w$ . For example, let  $w = b a^N b a^N$ . But then there are additional choices for what  $y$  could be (since  $y$  could include the initial  $b$ ) and we would have to work through them all.

(c)  $L = \{ww' : w \in \{a, b\}^*\}$ , where  $w'$  stands for  $w$  with each occurrence of  $a$  replaced by  $b$ , and vice versa. We can prove this easily using the pumping lemma. Let  $w = a^N b^N$ . Since  $|xy| \leq N$ ,  $y$  must be a string of all a's. So, when we pump (either in or out), we modify the first part of  $w$  but not the second part. Thus the resulting string is not in  $L$ .

We could also solve this problem just by observing that, if  $L$  is regular, so is  $L' = L \cap a^* b^*$ . But  $L'$  is just  $a^n b^n$ , which we have already shown is not regular. Thus  $L$  is not regular either.

2. (a)  $L = \{w : w \text{ is the unary notation for a natural number that is a multiple of } 7\}$ .  $L$  is regular since it can be described by the regular expression  $(1111111)^*$ .



(b)  $L = \{w : w \text{ is the decimal notation for a natural number that is a multiple of } 7\}$ .  $L$  is regular. We can build a deterministic FSM  $M$  to accept it.  $M$  is based on the standard algorithm for long division. The states represent the remainders we have seen so far (so there are 7 of them, corresponding to  $0 - 6$ ). The start state, of course, is 0, corresponding to a remainder of 0. So is the final state. The transitions of  $M$  are as follows:

$$\forall s_i \in \{0 - 6\} \text{ and } \forall c_j \in \{0 - 9\}, \delta(s_i, c_j) = (10s_i + c_j) \bmod 7$$

So, for example, on the input 962,  $M$  would first read 9. When you divide 7 into 9 you get 1 (which we don't care about since we don't actually care about the answer – we just care whether the remainder is 0) with a remainder of 2. So  $M$  will enter state 2. Next it reads 6. Since it is in state 2, it must divide 7 into  $2 \cdot 10 + 6$  (26). It gets a remainder of 5, so it goes to state 5. Next it reads 2. Since it is in state 5, it must divide 7 into  $5 \cdot 10 + 5$  (52), producing a remainder of 3. Since 3 is not zero, we know that 962 is not divisible by 7, so  $M$  rejects.

(c)  $L = \{w : w \text{ is the unary notation for a natural number such that there exists a pair } p \text{ and } q \text{ of twin primes, both } > n.\}$ .  $L$  is regular. Unfortunately, this time we don't know how to build a PDA for it. We can, however, prove that it is regular by considering the following two possibilities:

- (1) There is an infinite number of twin primes. In this case, for every  $n$ , there exists a pair of twin primes greater than  $n$ . Thus  $L = 1^*$ , which is clearly regular.
- (2) There is not an infinite number of twin primes. In this case, there is some largest pair. There is thus also a largest  $n$  that has a pair greater than it. Thus the set of such  $n$ 's is finite and so is  $L$  (the unary encodings of those values of  $n$ ). Since  $L$  is finite, it is clearly regular.

It is not known which of these cases is true. But interestingly, from our point of view, it doesn't matter.  $L$  is regular in either case. It may bother you that we can assert that  $L$  is regular when we cannot draw either an FSM or a regular expression for it. It shouldn't bother you. We have just given a nonconstructive proof that  $L$  is regular (and thus, by the way, that some FSM  $M$  accepts it). Not all proofs need to be constructive. This situation isn't really any different from the case of  $L' = \{w : w \text{ is the unary encoding of the number of siblings I have}\}$ . You know that  $L'$  is finite and thus regular, even though you do not know how many siblings I have and thus cannot actually build a machine to accept  $L'$ .

(d)  $L = \{w : w \text{ is, for some } n \geq 1, \text{ the unary notation for } 10^n\}$ . So  $L = \{1111111111, 1^{100}, 1^{1000}, \dots\}$ .  $L$  isn't regular, since clearly any machine to accept  $L$  will have to count the 1's. We can prove this using the pumping lemma: Let  $w = 1^P$ ,  $N \leq P$  and  $P$  is some power of 10.  $y$  must be some number of 1's. Clearly, it can be of length at most  $P$ . When we pump it in once, we get a string  $s$  whose maximum length is therefore  $2P$ . But the next power of 10 is  $10P$ . Thus  $s$  cannot be in  $L$ .

(e)  $L = \{w : w \text{ is, for some } n \geq 1, \text{ the decimal notation for } 10^n\}$ . Often it's easier to work with unary representations, but not in this case. This  $L$  is regular, since it is just  $100^*$ .

(f)  $L = \{w \text{ is of the form } x\#y, \text{ where } x, y \in \{1\}^+ \text{ and } y = x+1 \text{ when } x \text{ and } y \text{ are interpreted as unary numbers}\}$  (For example,  $11\#111$  and  $1111\#11111 \in L$ , while  $11\#11$ ,  $1\#111$ , and  $1111 \notin L$ .)  $L$  isn't regular. Intuitively, it isn't regular because any machine to accept it must count the 1's before the # and then compare that number to the number of 1's after the #. We can prove that this is true using the pumping lemma: Let  $w = 1^N\#1^{N+1}$ . Since  $|xy| \leq N$ ,  $y$  must occur in the region before the #. Thus when we pump (either in or out) we will change  $x$  but not make the corresponding change to  $y$ , so  $y$  will no longer equal  $x + 1$ . The resulting string is thus not in  $L$ .

(g)  $L = \{a^n b^j : |n - j| = 2\}$ .  $L$  isn't regular.  $L$  consists of all strings of the form  $a^*b^*$  where either the number of  $a$ 's is two more than the number of  $b$ 's or the number of  $b$ 's is two more than the number of  $a$ 's. We can show that  $L$  is not regular by pumping. Let  $w = a^N b^{N+2}$ . Since  $|xy| \leq N$ ,  $y$  must equal  $a^p$  for some  $p > 0$ . We can pump  $y$  out once, which will generate the string  $a^{N-p} b^{N+2}$ , which is not in  $L$ .

(h)  $L = \{uww^Rv : u, v, w \in \{a, b\}^+\}$ .  $L$  is regular. This may seem counterintuitive. But any string of length at least four with two consecutive symbols, not including the first and the last ones, is in  $L$ . We simply make everything up to the first of the two consecutive symbols  $u$ . The first of the two consecutive symbols is  $w$ . The second is  $w^R$ . And the rest of the string is  $v$ . And only strings with at least one pair of consecutive symbols (not including the first and last) are in  $L$  because  $w$  must end with some symbol  $s$ .  $w^R$  must start with that same symbol  $s$ . Thus the string will contain two consecutive occurrences of  $s$ .  $L$  is regular because it can be described the regular expression  $(a \cup b)^+ (aa \cup bb) (a \cup b)^+$ .

(i)  $L = \{w \in \{a, b\}^* : \text{for each prefix } x \text{ of } w, \#a(x) \geq \#b(x)\}$ . First we need to understand exactly what  $L$  is. In order to do that, we need to define prefix. A string  $x$  is a prefix of a string  $y$  iff  $\exists z \in \Sigma^*$  such that  $y = xz$ . In other words,  $x$  is a prefix of  $y$  iff  $x$  is an initial substring of  $y$ . For example, the prefixes of  $abba$  are  $\epsilon$ ,  $a$ ,  $ab$ ,  $abb$ , and  $abba$ . So  $L$  is all strings over  $\{a, b\}^*$  such that, at any point in the string (reading left to right), there have never been more  $b$ 's than  $a$ 's. The strings  $\epsilon$ ,  $a$ ,  $ab$ ,  $aaabbb$ , and  $ababa$  are in  $L$ . The strings  $b$ ,  $ba$ ,  $abba$ , and  $ababb$  are not in  $L$ .  $L$  is not regular, which we can show by pumping. Let  $w = a^N b^N$ . So  $y = a^p$ , for some nonzero  $p$ . If we pump out, there will be fewer  $a$ 's than  $b$ 's in the resulting string  $s$ . So  $s$  is not in  $L$  since every string is a prefix of itself.

3. (a) Every subset of a regular language is regular. FALSE. Often the easiest way to show that a universally quantified statement such as this is false by showing a counterexample. So consider  $L = a^*$ .  $L$  is clearly regular, since we have just shown a regular expression for it. Now consider  $L' = a^i : i \text{ is prime}$ .  $L' \subseteq L$ . But we showed in class that  $L'$  is not regular.

(b) Let  $L' = L1 \cap L2$ . If  $L'$  is regular and  $L2$  is regular,  $L1$  must be regular. FALSE. We know that the regular languages are closed under intersection. But it is important to keep in mind that this closure lemma (as well as all the others we will prove) only says exactly what it says and no more. In particular, it says that:

If  $L1$  is regular and  $L2$  is regular

Then  $L'$  is regular.

Just like any implication, we can't run this one backward and conclude anything from the fact that  $L'$  is regular. Of course, we can't use the closure lemma to say that  $L1$  must not be regular either. So we can't apply the closure lemma here at all. A rule of thumb: it is almost never true that you can prove the converse of a closure lemma. So it makes sense to look first for a counterexample. We don't have to look far. Let  $L' = \emptyset$ . Let  $L2 = \emptyset$ . So  $L'$  and  $L2$  are regular. Now let  $L1 = \{a^i : i \text{ is prime}\}$ .  $L1$  is not regular. Yet  $L' = L1 \cap L2$ . Notice that we could have made  $L2$  anything at all and its intersection with  $\emptyset$  would have been  $\emptyset$ . When you are looking for counterexamples, it usually works to look for very simple ones such as  $\emptyset$  or  $\Sigma^*$ , so it's a good idea to start there first.  $\emptyset$  works well in this case because we're doing intersection.  $\Sigma^*$  is often useful when we're doing union.

(c) If  $L$  is regular, then so is  $L' = \{xy : x \in L \text{ and } y \notin L\}$ . TRUE. Proof: Saying that  $y \notin L$  is equivalent to saying that  $y \in \overline{L}$ . Since the regular languages are closed under complement, we know that  $\overline{L}$  is also regular.  $L'$  is thus the concatenation of two regular languages. The regular languages are closed under concatenation. Thus  $L'$  must be regular.

(d)  $L = \{w : w = w^R\}$  is regular. FALSE.  $L$  is NOT regular. You can prove this easily by using the pumping lemma and letting  $w = a^N b a^N$ .

(e) If  $L$  is a regular language, then so is  $L' = \{w : w \in L \text{ and } w^R \in L\}$ . TRUE. Proof: Saying that  $w^R \in L$  is equivalent to saying that  $w \in L^R$ . If  $w$  must be in both  $L$  and  $L^R$ , that is equivalent to saying that  $L' = L \cap L^R$ .  $L$  is regular because the problem statement says so.  $L^R$  is also regular because the regular languages are closed

under reversal. The regular languages are closed under intersection. So the intersection of  $L$  and  $L^R$  must be regular.

Proof that the regular languages are closed under reversal (by construction): If  $L$  is regular, then there exists some FSM  $M$  that accepts it. From  $M$ , we can construct a new FSM  $M'$  that accepts  $L^R$ .  $M'$  will effectively run  $M$  backwards. Start with the states of  $M'$  equal to states of  $M$ . Take the state that corresponds to the start state of  $M$  and make it the final state of  $M'$ . Next we want to take the final states of  $M$  and make them the start states of  $M'$ . But  $M'$  can have only a single start state. So create a new start state in  $M'$  and create an epsilon transition from it to each of the states in  $M'$  that correspond to final states of  $M$ . Now just flip the arrows on all the transitions of  $M$  and add these new transitions to  $M'$ .

(f) If  $C$  is any set of regular languages,  $\cup C$  is a regular language. FALSE. If  $C$  is a *finite* set of regular languages, this is true. It follows from the fact that the regular languages are closed under union. But suppose that  $C$  is an infinite set of languages. Then this statement cannot be true. If it were, then every language would be regular and we have proved that there are languages that are not regular. Why is this? Because every language is the union of some set of regular languages. Let  $L$  be an arbitrary language whose elements are  $w_1, w_2, w_3, \dots$ . Let  $C$  be the set of singleton languages  $\{\{w_1\}, \{w_2\}, \{w_3\}, \dots\}$  such that  $w_i \in L$ . The number of elements of  $C$  is equal to the cardinality of  $L$ . Each individual element of  $C$  is a language that contains a single string, and so it is finite and thus regular.  $L = \cup C$ . Thus, since not all languages are regular, it must not be the case that  $\cup C$  is guaranteed to be regular. If you're not sure you follow this argument, you should try to come up with a specific counterexample. Choose an  $L$  such that  $L$  is not regular, and show that it can be described as  $\cup C$  for some set of languages  $C$ .

(g)  $L = \{xyx^R : x, y \in \Sigma^*\}$  is regular. TRUE. Why? We've already said that  $xx^R$  isn't regular. This looks a lot like that, but it differs in a key way.  $L$  is the set of strings that can be described as some string  $x$ , followed by some string  $y$  (where  $x$  and  $y$  can be chosen completely independently), followed by the reverse of  $x$ . So, for example, it is clear that  $abccccba \in L$  (assuming  $\Sigma = \{a, b, c\}$ ). We let  $x = ab$ ,  $y = ccccc$ , and  $x^R = ba$ . Now consider  $abbccccaaa$ . You might think that this string is not in  $L$ . But it is. We let  $x = a$ ,  $y = bbccccaa$ , and  $x^R = a$ . What about  $accb$ ? This string too is in  $L$ . We let  $x = \epsilon$ ,  $y = accb$ , and  $x^R = \epsilon$ . Note the following things about our definition of  $L$ : (1) There is no restriction on the length of  $x$ . Thus we can let  $x = \epsilon$ . (2) There is no restriction on the relationship of  $y$  to  $x$ . And (3)  $\epsilon^R = \epsilon$ . Thus  $L$  is in fact equal to  $\Sigma^*$  because we can take any string  $w$  in  $\Sigma^*$  and rewrite it as  $\epsilon w \epsilon$ , which is of the form  $xyx^R$ . Since  $\Sigma^*$  is regular,  $L$  must be regular.

(h) If  $L' = L1 \cup L2$  is a regular language and  $L1$  is a regular language, then  $L2$  is a regular language. FALSE. This is another attempt to use a closure theorem backwards. Let  $L1 = \Sigma^*$ .  $L1$  is clearly regular. Since  $L1$  contains all strings over  $\Sigma$ , the union of  $L1$  with any language is just  $L1$  (i.e.,  $L' = \Sigma^*$ ). If the proposition were true, then all languages  $L2$  would necessarily be regular. But we have already shown that there are languages that are not regular. Thus the proposition must be false.

(i) Every regular language has a regular proper subset. FALSE.  $\emptyset$  is regular. And it is subset of every set. Thus it is a subset of every regular language. However, it is not a *proper* subset of itself. Thus this statement is false. However the following two similar statements are true:

- (1) Every regular language has a regular subset.
- (2) Every regular language except  $\emptyset$  has a regular proper subset.

(j) If  $L1$  and  $L2$  are nonregular languages, then  $L1 \cup L2$  is also not regular. False. Let  $L1 = \{a^n b^m, n \geq m\}$  and  $L2 = \{a^n b^m, n \leq m\}$ .  $L1 \cup L2 = a^* b^*$ , which is regular.

4. If  $L$  were regular, then its complement,  $L_1$ , would also be regular.  $L_1$  contains all strings over  $\{a, b\}$  that are not in  $L$ . There are two ways not to be in  $L$ : have any a's that occur after any b's (in other words, not have all the a's followed by all the b's), or have an equal number of a's and b's. So now consider

$$L_2 = L_1 \cap a^*b^*$$

$L_2$  contains only those elements of  $L_1$  in which the a's and b's are in the right order. In other words,

$$L_2 = a^n b^n$$

But if  $L$  were regular, then  $L_1$  would be regular. Then  $L_2$ , since it is the intersection of two regular languages would also be regular. But we have already shown that it ( $a^n b^n$ ) is not regular. Thus  $L$  cannot be regular.

5. This statement is false. To prove it, we offer a counter example. Let  $L_1 = \{a^n b^m : n=m\}$  and let  $L_2 = \{a^n b^m : n \neq m\}$ . We have shown that both  $L_1$  and  $L_2$  are not regular. However,

$$L_1 \cup L_2 = a^*b^*, \text{ which is regular.}$$

There are plenty of other examples as well. Let  $L_1 = \{a^n : n \geq 1 \text{ and } n \text{ is prime}\}$ . Let  $L_2 = \{a^n : n \geq 1 \text{ and } n \text{ is not prime}\}$ . Neither  $L_1$  nor  $L_2$  is regular. But  $L_1 \cup L_2 = a^+$ , which is clearly regular.

6. This is easy to prove using the pumping lemma. Let  $w = a^N b a^N b a^N$ . We know that  $xy$  must be contained within the first block of a's. So, no matter how  $y$  is chosen (as long as it is not empty, as required by the lemma), for any  $i > 2$ ,  $xy^i z \notin L$ , since the first block of a's will be longer than the last block, which is not allowed. Therefore  $L$  is not regular.

7. First, let  $L' = L \cap a^*b^*$ , which must be regular if  $L$  is. We observe that  $L' = a^n b^{n+2} : n \geq 0$ . Now use the pumping lemma to show that  $L'$  is not regular in the same way we used it to show that  $a^n b^n$  is not regular.

8. We use the pumping lemma. Let  $w = a^{2N} b^N$ .  $xy$  must be contained within the block of a's, so when we pump either in or out, it will no longer be true that there will be twice as many a's as b's, since the number of a's changes but not the number of b's. Thus the pumped string will not be in  $L$ . Therefore  $L$  is not regular.

9. (a)  $L$  is not regular. We can prove this using the pumping lemma. Let  $w = a^N b^N$ . Since  $y$  must occur within the first  $N$  characters of  $w$ ,  $y = a^p$  for some  $p > 0$ . Thus when we pump  $y$  in, we will have more a's than b's, which produces strings that are not in  $L$ .

(b)  $L^*$  is also not regular. To prove this, we need first to prove a lemma, which we'll call EQAB:  $\forall s, s \in L^* \Rightarrow \#a(s) = \#b(s)$ . To prove the lemma, we first observe that any string  $s$  in  $L^*$  must be able to be decomposed into at least one finite sequence of strings, each element of which is in  $L$ . Some strings will have multiple such decompositions. In other words, there may be more than one way to form  $s$  by concatenating together strings in  $L$ . For any string  $s$  in  $L^*$ , let  $SQ$  be some sequence of elements of  $L$  that, when concatenated together, form  $s$ . It doesn't matter which one. Define the function  $\text{HowMany}$  on the elements of  $L^*$ .  $\text{HowMany}(x)$  returns the length of  $SQ$ . Think of  $\text{HowMany}$  as telling you how many times we went through the Kleene star loop in deriving  $x$ . We will prove EQAB by induction on  $\text{HowMany}(s)$ .

Base case: If  $\text{HowMany}(s) = 0$ , then  $s = \epsilon$ .  $\#a(s) = \#b(s)$ .

Induction hypothesis: If  $\text{HowMany}(s) \leq N$ , then  $\#a(s) = \#b(s)$ .

Show: If  $\text{HowMany}(s) = N+1$ , then  $\#a(s) = \#b(s)$ .

If  $\text{HowMany}(s) = N+1$ , then  $\exists w, y$  such that  $s = wy$ ,  $w \in L^*$ ,  $\text{HowMany}(w) = N$ , and  $y \in L$ . In other words, we can decompose  $s$  into a part that was formed by concatenating together  $N$  instances of  $L$  plus a second part that is just one more instance of  $L$ . Thus we have:

(1) $\#a(y) = \#b(y)$ .	Definition of L	
(2) $\#a(w) = \#b(w)$ .	Induction hypothesis	
(3) $\#a(s) = \#a(w) + \#a(y)$	$s = wy$	
(4) $\#b(s) = \#b(w) + \#b(y)$ .	$s = wy$	
(5) $\#b(s) = \#a(w) + \#b(y)$	4, 2	
(6) $\#b(s) = \#a(w) + \#a(y)$	5, 1	
(7) $\#b(s) = \#a(s)$	6, 3	Q. E. D.

Now we can show that  $L^*$  isn't regular using the pumping lemma. Let  $w = a^N b^N$ . Since  $y$  must occur within the first  $N$  characters of  $w$ ,  $y = a^p$  for some  $p > 0$ . Thus when we pump  $y$  in, we will have a string with more  $a$ 's than  $b$ 's. By EQAB, that string cannot be in  $L^*$ .

## CS 341 Homework 10 State Minimization

1. (a) Give the equivalence classes under  $\approx_L$  for these languages:

- (i)  $L = (aab \cup ab)^*$
- (ii)  $L = \{x : x \text{ contains an occurrence of } aababa\}$
- (iii)  $L = \{xx^R : x \in \{a, b\}^*\}$
- (iv)  $L = \{xx : x \in \{a, b\}^*\}$
- (v)  $L_n = \{a, b\}a\{a, b\}^n$ , where  $n > 0$  is a fixed integer
- (vi) The language of balanced parentheses

(b) For those languages in (a) for which the answer is finite, give a deterministic finite automaton with the smallest number of states that accepts the corresponding language.

2. Let  $L = \{x \in \{a, b\}^* : x \text{ contains at least one } a \text{ and ends in at least two } b\text{'s}\}$ .

- (a) Write a regular expression for  $L$ .
- (b) Construct a deterministic FSM that accepts  $L$ .
- (c) Let  $R_L$  be the equivalence relation of the Myhill-Nerode Theorem. What partition does  $R_L$  induce on the set  $\{a, bb, bab, abb, bba, aab, abba, bbaa, baaba\}$ ?
- (d) How many equivalence classes are there in the partition induced on  $\Sigma^*$  by  $R_L$ ?

3. Let  $L = \{x \in \{a, b\}^* : x \text{ begins with } a \text{ and ends with } b\}$ .

- (a) What is the nature of the partition induced on  $\Sigma^*$  by  $R_L$ , the equivalence relation of the Myhill-Nerode Theorem? That is, how many classes are there in the partition and give a description of the strings in each.
- (b) Using these equivalence classes, construct the minimum state deterministic FSM that accepts  $L$ .

4. Suppose that we are given a language  $L$  and a deterministic FSM  $M$  that accepts  $L$ . Assume  $L$  is a subset of  $\{a, b, c\}^*$ . Let  $R_L$  and  $R_M$  be the equivalence relations defined in the proof of the Myhill-Nerode Theorem. True or False:

- (a) If we know that  $x$  and  $y$  are two strings in the same equivalence class of  $R_L$ , we can be sure that they are in the same equivalence class of  $R_M$ .
- (b) If we know that  $x$  and  $y$  are two strings in the same equivalence class of  $R_M$ , we can be sure that they are in the same equivalence class of  $R_L$ .
- (c) There must be at least one equivalence class of  $R_L$  that has contains an infinite number of strings.
- (d)  $R_M$  induces a partition on  $\{a, b, c\}^*$  that has a finite number of classes.
- (e) If  $\varepsilon \in L$ , then  $[\varepsilon]$  (the equivalence class containing  $\varepsilon$ ) of  $R_L$  cannot be an infinite set.

5. Use the Myhill-Nerode Theorem to prove that  $\{a^n b^m c^{\max(m,n)} b^p d^p : m, n, p \geq 0\}$  is not regular.

6. (a) In class we argued that the intersection of two regular languages was regular on the basis of closure properties of regular languages. We did not show a construction for the FSM that recognizes the intersection of two regular languages. Such a construction does exist, however, and it is suggested by the fact that  $L_1 \cap L_2 = \Sigma^* - ((\Sigma^* - L_1) \cup (\Sigma^* - L_2))$ .

Given two deterministic FSMs,  $M_1$  and  $M_2$ , that recognize two regular languages  $L_1$  and  $L_2$ , we can construct an FSM that recognizes  $L = L_1 \cap L_2$  (in other words strings that have all the required properties of both  $L_1$  and  $L_2$ ), as follows:

1. Construct machines  $M_1'$  and  $M_2'$ , as deterministic versions of  $M_1$  and  $M_2$ . This step is necessary because complementation only works on deterministic machines.
2. Construct machines  $M_1''$  and  $M_2''$ , from  $M_1'$  and  $M_2'$ , using the construction for complementation, that recognize  $\Sigma^* - L_1$  and  $\Sigma^* - L_2$ , respectively.
3. Construct  $M_3$ , using the construction for union and the machines  $M_1''$  and  $M_2''$ , that recognizes  $(\Sigma^* - L_1) \cup (\Sigma^* - L_2)$ . This will be a nondeterministic FSM.
4. Construct  $M_4$ , the deterministic equivalent of  $M_3$ .
5. Construct  $M_L$ , using the construction for complementation, that recognizes  $\Sigma^* - ((\Sigma^* - L_1) \cup (\Sigma^* - L_2))$ .

Now consider:

$$\Sigma = \{a, b\}$$

$$L_1 = \{w \in \Sigma^* : \text{all } a\text{'s occur in pairs}\}$$

$$\text{e.g., } aa, aaaa, aabaa, aabbaabbb \in L_1$$

$$aaa, baaab, ab \notin L_1$$

$$L_2 = \{w \in \Sigma^* : w \text{ contains the string } bbb\}$$

Use the procedure outlined above to construct an FSM  $M_L$  that recognizes  $L = L_1 \cap L_2$ .

Is  $M_L$  guaranteed to be deterministic?

(b) What are the equivalence classes under  $\approx_L$  for the language  $L = L_1 \cap L_2$ ?

(c) What are the equivalence classes under  $\sim_M$  for  $M_L$  in (a) above?

(d) Show how  $\sim_M$  is a refinement of  $\approx_L$ .

(e) Use the minimization algorithm that we have discussed to construct from  $M_L$  in (a) above a minimal state machine that accepts  $L$ .

7. If you had trouble with this last one, make up another pair of  $L_1$  and  $L_2$  and try again.

## Solutions

### 1. (a)

(i)  $L = (aab \cup ab)^*$

1.  $[\epsilon, aab, ab, \text{ and all other elements of } L]$
2.  $[a \text{ or } wa : w \in L]$
3.  $[aa \text{ or } waa : w \in L]$
4. [everything else, i.e., strings that can never become elements of  $L$  because they contain illegal substrings such as  $bb$  or  $aaa$ ]

(ii)  $L = \{x : x \text{ contains an occurrence of } aababa\}$

1.  $[(a \cup b)^* aababa (a \cup b)^*, \text{ i.e., all elements of } L]$
2.  $[\epsilon \text{ or any string not in } L \text{ and ending in } b \text{ but not in } aab \text{ or } aabab, \text{ i.e., no progress yet on "aababa"}]$
3.  $[wa \text{ for any } w \in [2]; \text{ alternatively, any string not in } L \text{ and ending in } a \text{ but not in } aa, aaba, \text{ or } aababa]$
4. [any string not in  $L$  and ending in  $aa$ ]
5. [any string not in  $L$  and ending in  $aab$ ]
6. [any string not in  $L$  and ending in  $aaba$ ]
7. [any string not in  $L$  and ending in  $aabab$ ]

Note that this time there is no "everything else". Strings never become hopeless in this language. They simply fail if we get to the end without finding "aababa".

(iii)  $L = \{xx^R : x \in \{a, b\}^*\}$

1.  $[a, \text{ which is the only string for which the continuations that lead to acceptance are all strings of the form } wa : \text{ where } w \in L]$
2.  $[b, \text{ which is the only string for which the continuations that lead to acceptance are all strings of the form } wb : \text{ where } w \in L]$
3.  $[ab, \text{ which is the only string for which the continuations that lead to acceptance are all strings of the form } wba : \text{ where } w \in L]$
4.  $[aa, \text{ which is the only string for which the continuations that lead to acceptance are all strings of the form } waa : \text{ where } w \in L]$

And so forth. Every string is in a distinct equivalence class.

(iv)  $L = \{xx : x \in \{a, b\}^*\}$

1.  $[a, \text{ which is the only string for which the continuations that lead to acceptance are all strings that would be in } L \text{ except that they are missing a leading } a]$
2.  $[b, \text{ which is the only string for which the continuations that lead to acceptance are all strings that would be in } L \text{ except that they are missing a leading } b]$
3.  $[ab, \text{ which is the only string for which the continuations that lead to acceptance are all strings that would be in } L \text{ except that they are missing a leading } ab]$

4. [aa, which is the only string for which the continuations that lead to acceptance are all strings that would be in L except that they are missing a leading aa]

And so forth. Every string is in a distinct equivalence class.

(v)  $L_n = \{a, b\}a\{a, b\}^n$

0. [ε]
1. [a, b]
2. [aa, ba]
3. [aaa, aab, baa, bab]

⋮  
⋮

n+2. [(a ∪ b)a(a ∪ b)<sup>n</sup>]

n+3. [strings that can never become elements of L<sub>n</sub>]

There is a finite number of strings in any specific language L<sub>n</sub>. So there is a finite number of equivalence classes of  $\approx_L$ . Every string in L<sub>n</sub> must be of length n+2. So there are n+3 equivalence classes (numbered 0 to n+2, to indicate the length of the strings in the class) of strings that may become elements of L<sub>n</sub>, plus one for strings that are already hopeless, either because they don't start with ab or aa, or because they are already too long.

(vi) L = The language of balanced parentheses

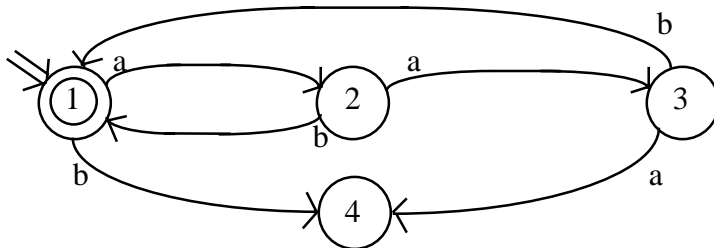
1. [w\*(w\*: w ∈ L) /\* i.e., one extra left parenthesis somewhere in the string
2. [w\*((w\*: w ∈ L) /\* two “
3. [w\*(((w\*: w ∈ L)
4. [w\*(((w\*: w ∈ L)
5. [w\*(((w\*: w ∈ L)

... and so on. There is an infinite number of equivalence classes.

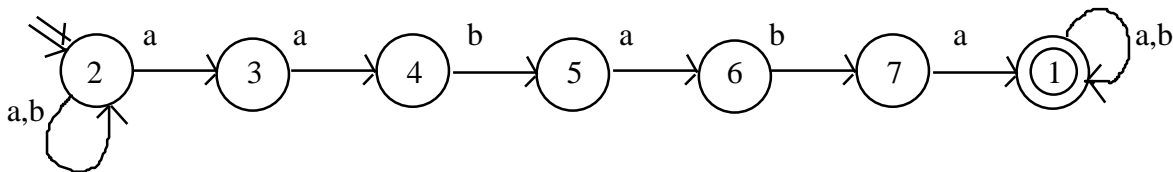
Each of these classes is distinct, since ) is an acceptable continuation for 1, but none of the others; )) is acceptable for 2, but none of the others, ))) is acceptable for 3, but none of the others, and so forth.

**1. (b)**

(i)

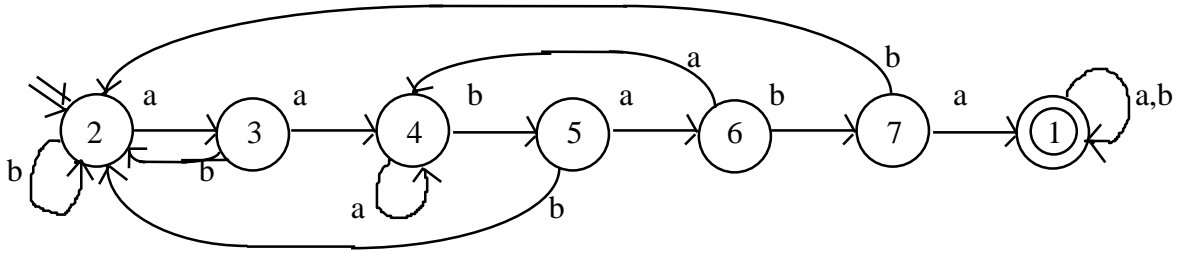


(ii) There's always a very simple nondeterministic FSM to recognize all strings that contain a specific substring. It's just a chain of states for the desired substring, with loops on all letters of Σ on the start state and the final state. In this case, the machine is:



To construct a minimal, deterministic FSM, you have two choices. You can use our algorithm to convert the NDFSM to a deterministic one and then use the minimization algorithm to get the minimal machine. Or you can construct the minimal FSM directly. In any case, it is:

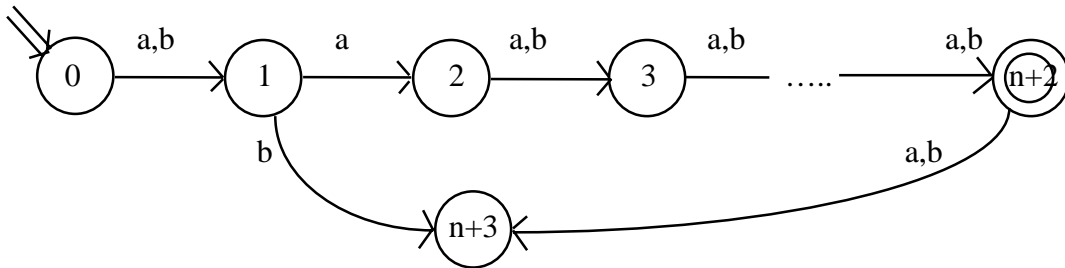




(iii) There is no FSM for this language, since it is not regular.

(iv) There is no FSM for this language, since it is not regular.

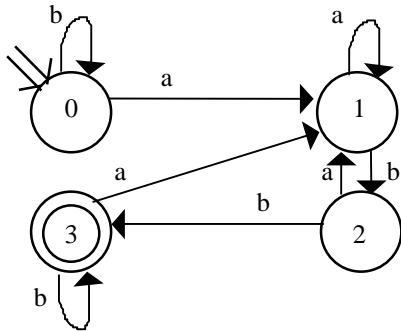
(v)



(vi) There is no FSM for this language, since it is not regular.

2. (a)  $(a \cup b)^* a (a \cup b)^* b b b^*$  or  $(a \cup b)^* a (a \cup b)^* b b$

(b)



(c) It's easiest to answer (d) first, and then to consider (c) as a special case.

(d) [0] = all strings that contain no a

[1] = all strings that end with a

[2] = all strings that end with ab

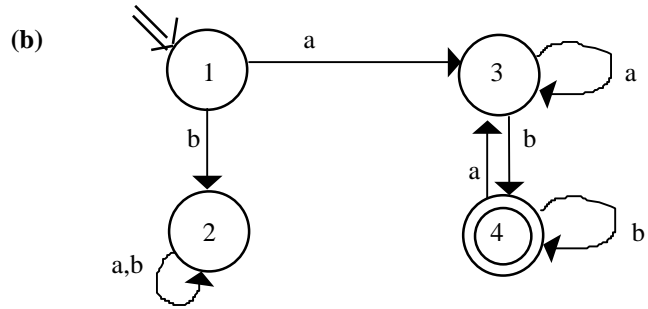
[3] = all strings that contain at least one a and that end with bb, i.e., all strings in L.

It is clear that these classes are pairwise disjoint and that their union is  $\{a, b\}^*$ . Thus they represent a partition of  $\{a, b\}^*$ . It is also easy to see that they are the equivalence classes of  $R_L$  of the Myhill-Nerode Theorem, since all the members of one equivalence class will, when suffixed by any string  $z$ , form strings all of which are in L or all of which are not in L. Further, for any  $x$  and  $y$  from different equivalence classes, it is easy to find a  $z$  such that one of  $xz, yz$  is in L and the other is not.

Letting the equivalence relation  $R_L$  be restricted to the set in part (c), gives the partition

$\{\{bb\}, \{a, bba, abba, bbaa, baaba\}, \{bab, aab\}, \{abb\}\}$ .

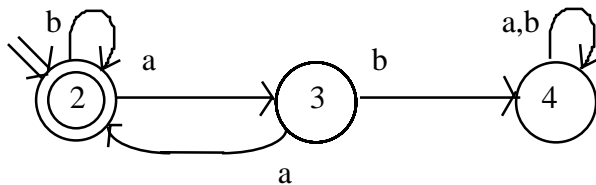
3. (a) [1] = { $\epsilon$ }  
 [2] =  $b(a \cup b)^*$   
 [3] =  $a \cup a(a \cup b)^*a$   
 [4] =  $a(a \cup b)^*b$



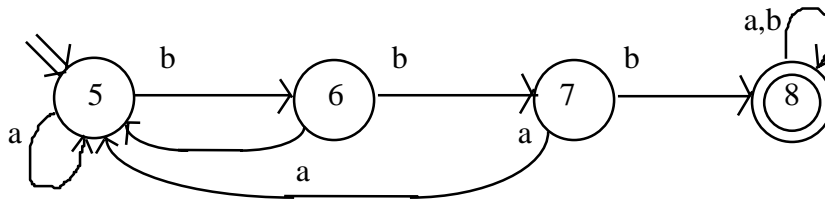
4. (a) F, (b) T, (c) T ( $\Sigma^*$  is infinite and the number of equivalence classes is finite), (d) T, (e) F.

5. Choose any two distinct strings of a's: call them  $a^i$  and  $a^j$  ( $i < j$ ). Then they must be in different equivalence classes of  $R_L$  since  $a^i b^i c^i \in L$  but  $a^j b^i c^i \notin L$ . Therefore, there is an infinite number of equivalence classes and  $L$  is not regular.

6. (a)  $M_1$ , which recognizes  $L_1$ , is:



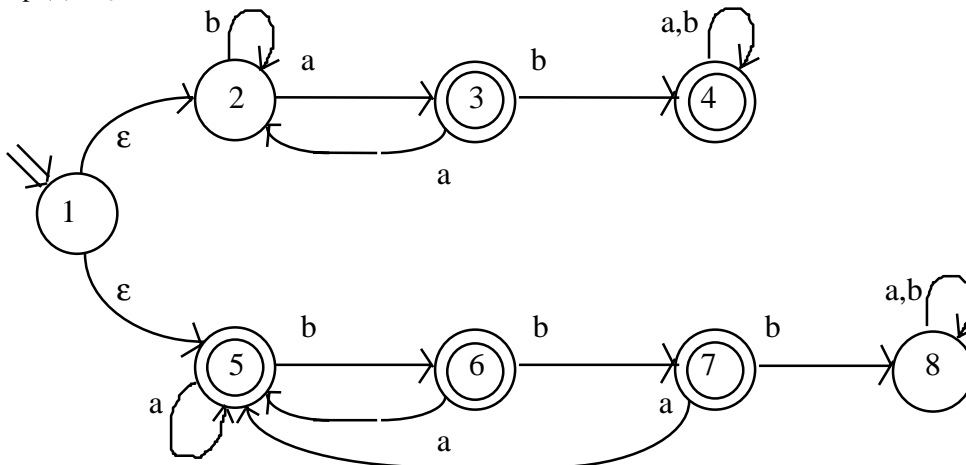
$M_2$ , which recognizes  $L_2$ , is:



- Step (1)  $M_1' = M_1$  because  $M_1$  is deterministic.  
 $M_2' = M_2$  because  $M_2$  is deterministic.

- Step (2)  $M_1''$  is  $M_1'$  except that states 3 and 4 are the final states.  
 $M_2''$  is  $M_2'$  except that states 5, 6, and 7 are the final states.

- Step (3)  $M_3$  is:



Step (4)  $M_4$  is:

$\{1, 2, 5\}, a, \{3, 5\}$ b, $\{2, 6\}$	$\{2, 5\}, a, \{3, 5\}$ b, $\{2, 6\}$	$\{4, 5\}, a, \{4, 5\}$ b, $\{4, 6\}$	$\{4, 8\}, a, \{4, 8\}$ b, $\{4, 8\}$
$\{3, 5\}, a, \{2, 5\}$ b, $\{4, 6\}$	$\{4, 6\}, a, \{4, 5\}$ b, $\{4, 7\}$	$\{4, 7\}, a, \{4, 5\}$ b, $\{4, 8\}$	$\{3, 8\}, a, \{2, 8\}$ b, $\{4, 8\}$
$\{2, 6\}, a, \{3, 5\}$ b, $\{2, 7\}$	$\{2, 7\}, a, \{3, 5\}$ b, $\{2, 8\}$	$\{2, 8\}, a, \{3, 8\}$ b, $\{2, 8\}$	

$s = \{1, 2, 5\}$

$F = K - \{2, 8\}$ , i.e., all states except  $\{2, 8\}$  are final states.

You may find it useful at this point to draw this out.

Step (5)  $M_L = M_4$  except that now there is a single final state,  $\{2, 8\}$ .

$M_L$  is deterministic.  $M_4$  is deterministic by construction, and Step 5 can not introduce any nondeterminism since it doesn't introduce any transitions.

- (b)
- [strings without bbb and with any a's in pairs, including  $\epsilon$ ]
  - [strings without bbb but with a single a at the end]
  - [strings that cannot be made legal because they have a single a followed by a b]
  - [strings without bbb, with any a's in pairs, and ending in a single b]
  - [strings without bbb, with any a's in pairs, and ending with just two b's]
  - [strings with bbb and with any a's in pairs]
  - [strings with bbb but with a single a at the end]

- (c)
- |               |   |
|---------------|---|
| $\{1, 2, 5\}$ | $[\epsilon]$  |
| $\{3, 5\}$    | [strings without bbb but with a single a at the end]  |
| $\{2, 6\}$    | [strings without bbb, with any a's in pairs, and ending in a single b]  |
| $\{2, 5\}$    | [strings without bbb and with at least one pair of a's and any a's in pairs]  |
| $\{4, 6\}$    | [strings that cannot be made legal because they have a single a followed by a b and where every b is preceded by an a and the last character is b]                  |
| $\{2, 7\}$    | [strings without bbb, with any a's in pairs, and ending with just two b's]  |
| $\{4, 5\}$    | [strings that cannot be made legal because they have a single a followed by b and where there is no bbb and the last character is a]                                |
| $\{4, 7\}$    | [strings that cannot be made legal because they have a single a followed by a b and where there is no bbb but there is at least one bb and the last character is b] |
| $\{2, 8\}$    | [all strings in L]  |
| $\{4, 8\}$    | [strings that cannot be made legal because they have a single a followed by a b and where there is a bbb, but the ab violation came before the first bbb]           |
| $\{3, 8\}$    | [strings with bbb but with a single a at the end]   |

- (d)
- $$\{1, 2, 5\} \cup \{2, 5\} = 1$$
- $$\{3, 5\} = 2$$
- $$\{4, 6\} \cup \{4, 5\} \cup \{4, 7\} \cup \{4, 8\} = 3$$
- $$\{2, 6\} = 4$$
- $$\{2, 7\} = 5$$
- $$\{2, 8\} = 6$$
- $$\{3, 8\} = 7$$

(e)  $\equiv_0 = A: \{\{2, 8\}\}$ ,

B:  $\{\{1, 2, 5\}, \{2, 5\}, \{3, 5\}, \{4, 6\}, \{4, 5\}, \{4, 7\}, \{4, 8\}, \{2, 6\}, \{2, 7\}, \{3, 8\}\}$

To compute  $\equiv_1$ : Consider B (since clearly A cannot be split). We need to look at all the single character transitions out of each of these states. We've already done that in Step (3) of part (a) above, so we can use that table to tell us which of our current states each state goes to. Now we just need to use that to determine which element of  $\equiv_0$  they go to. We notice that all transitions are to elements of B except:  $(\{2, 7\}, b, A)$  and  $(\{3, 8\}, a, A)$ . So we must split these two states from B. and they must be distinct from each other because their a and b behaviors are reversed. So we have:

$\equiv_1 = A: \{\{2, 8\}\}$ ,

B:  $\{\{1, 2, 5\}, \{2, 5\}, \{3, 5\}, \{4, 6\}, \{4, 5\}, \{4, 7\}, \{4, 8\}, \{2, 6\}\}$

C:  $\{\{2, 7\}\}$

D:  $\{\{3, 8\}\}$

To compute  $\equiv_2$ : Again we consider B:

On reading an a, all elements of B go to elements of B.

But on b:  $(\{2, 6\}, b, C)$ , so we must split off  $\{2, 6\}$ . This gives us:

$\equiv_2 = A: \{\{2, 8\}\}$ ,

B:  $\{\{1, 2, 5\}, \{2, 5\}, \{3, 5\}, \{4, 6\}, \{4, 5\}, \{4, 7\}, \{4, 8\}\}$

C:  $\{\{2, 7\}\}$

D:  $\{\{3, 8\}\}$

E:  $\{\{2, 6\}\}$

To compute  $\equiv_3$ : Again we consider B:

On reading an a, all elements of B go to elements of B.

But on b,  $\{1, 2, 5\}$  and  $\{2, 5\}$  go to E, while everyone else goes to B. So we have to split these two off. This gives us:

$\equiv_3 = A: \{\{2, 8\}\}$ ,

B:  $\{\{3, 5\}, \{4, 6\}, \{4, 5\}, \{4, 7\}, \{4, 8\}\}$

C:  $\{\{2, 7\}\}$

D:  $\{\{3, 8\}\}$

E:  $\{\{2, 6\}\}$

F:  $\{\{1, 2, 5\}, \{2, 5\}\}$

To compute  $\equiv_4$ : Again we consider B:

On reading b, all elements of B stay in B. But on reading a,  $\{3, 5\}$  goes to F, so we split it off. This gives us:

$\equiv_4 = A: \{\{2, 8\}\}$ ,

B:  $\{\{4, 6\}, \{4, 5\}, \{4, 7\}, \{4, 8\}\}$

C:  $\{\{2, 7\}\}$

D:  $\{\{3, 8\}\}$

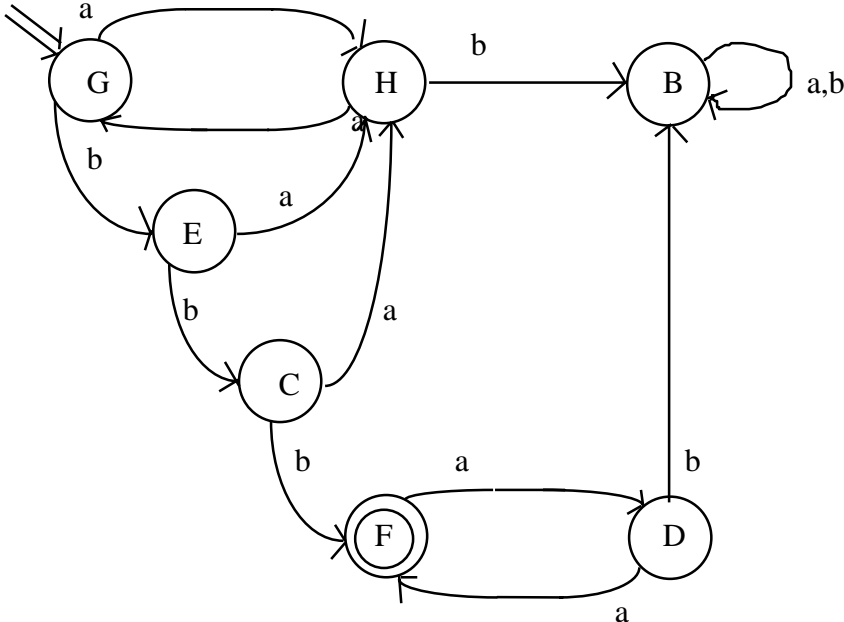
E:  $\{\{2, 6\}\}$

F:  $\{\{1, 2, 5\}, \{2, 5\}\}$

G:  $\{\{3, 5\}\}$

To compute  $\equiv_5$ : Again we consider B: On both inputs, all elements of B stay in B. So we do no further splitting, and we assert that  $\equiv = \equiv_4$ . Notice that this is identical to what we expected from (d) above.

We can now show the minimal machine:



## CS 341 Homework 11

### Context-Free Grammars

1. Consider the grammar  $G = (V, \Sigma, R, S)$ , where

$$\begin{aligned} V &= \{a, b, S, A\}, \\ \Sigma &= \{a, b\}, \\ R &= \{ S \rightarrow AA, \\ &\quad A \rightarrow AAA, \\ &\quad A \rightarrow a, \\ &\quad A \rightarrow bA, \\ &\quad A \rightarrow Ab \}. \end{aligned}$$

- (a) Which strings of  $L(G)$  can be produced by derivations of four or fewer steps?
- (b) Give at least four distinct derivations for the string babbab.
- (c) For any  $m, n, p > 0$ , describe a derivation in  $G$  of the string  $b^m a b^n a b^p$ .

2. Construct context-free grammars that generate each of these languages:

- (a)  $\{wcw^R : w \in \{a, b\}^*\}$
- (b)  $\{ww^R : w \in \{a, b\}^*\}$
- (c)  $\{w \in \{a, b\}^* : w = w^R\}$

3. Consider the alphabet  $\Sigma = \{a, b, (, ), \cup, *, \emptyset\}$ . Construct a context-free grammar that generates all strings in  $\Sigma^*$  that are regular expressions over  $\{a, b\}$ .

4. Let  $G$  be a context-free grammar and let  $k > 0$ . We let  $L_k(G) \subseteq L(G)$  be the set of all strings that have a derivation in  $G$  with  $k$  or fewer steps.

- (a) What is  $L_5(G)$ , where  $G = (\{S, (, )\}, \{(, )\}, \{S \rightarrow \epsilon, S \rightarrow SS, S \rightarrow (S)\})$ ?
- (b) Show that, for all context-free grammars  $G$  and all  $k > 0$ ,  $L_k(G)$  is finite.

5. Let  $G = (V, \Sigma, R, S)$ , where

$$\begin{aligned} V &= \{a, b, S\}, \\ \Sigma &= \{a, b\}, \\ R &= \{ S \rightarrow aSb, \\ &\quad S \rightarrow aSa, \\ &\quad S \rightarrow bSa, \\ &\quad S \rightarrow bSb, \\ &\quad S \rightarrow \epsilon \}. \end{aligned}$$

Show that  $L(G)$  is regular.

6. A program in a procedural programming language, such as C or Java, consists of a list of statements, where each statement is one of several types, such as:

- (1) assignment statement, of the form  $\text{id} := E$ , where  $E$  is any arithmetic expression (generated by the grammar using  $T$  and  $F$  that we presented in class).
- (2) conditional statement, e.g., "if  $E < E$  then statement", or while statement, e.g. "while  $E < E$  do statement".
- (3) goto statement; furthermore each statement could be preceded by a label.
- (4) compound statement, i.e., many statements preceded by a begin, followed by an end, and separated by ";".

Give a context-free grammar that generates all possible statements in the simplified programming language described above.

7. Show that the following languages are context free by exhibiting context-free grammars generating each:

- (a)  $\{a^m b^n : m \geq n\}$

- (b)  $\{a^m b^n c^p d^q : m + n = p + q\}$
- (c)  $\{w \in \{a, b\}^* : w \text{ has twice as many b's as a's}\}$
- (d)  $\{uawb : u, w \in \{a, b\}^*, |u| = |w|\}$

8. Let  $\Sigma = \{a, b, c\}$ . Let L be the language of prefix arithmetic defined as follows:

- (i) any member of  $\Sigma$  is a well-formed expression (wff).
- (ii) if  $\alpha$  and  $\beta$  are any wff's, then so are  $A\alpha\beta$ ,  $S\alpha\beta$ ,  $M\alpha\beta$ , and  $D\alpha\beta$ .
- (iii) nothing else is a wff.

(One might think of A, S, M, and D as corresponding to the operators +, -,  $\times$ , /, respectively. Thus in L we could write Aab instead of the usual (a + b), and MSabDbc, instead of  $((a - b) \times (b/c))$ . Note that parentheses are unnecessary to resolve ambiguities in L.)

- (a) Write a context-free grammar that generates exactly the wff's of L.
- (b) Show that L is not regular.

9. Consider the language  $L = \{a^m b^{2n} c^{3n} d^p : p > m, \text{ and } m, n \geq 1\}$ .

- (a) What is the shortest string in L?
- (b) Write a context-free grammar to generate L.

### Solutions

1. (a) We can do an exhaustive search of all derivations of length no more than 4:

- $S \Rightarrow AA \Rightarrow aA \Rightarrow aa$
- $S \Rightarrow AA \Rightarrow aA \Rightarrow abA \Rightarrow aba$
- $S \Rightarrow AA \Rightarrow aA \Rightarrow aAb \Rightarrow aab$
- $S \Rightarrow AA \Rightarrow bAA \Rightarrow baA \Rightarrow baa$
- $S \Rightarrow AA \Rightarrow bAA \Rightarrow bAa \Rightarrow baa$
- $S \Rightarrow AA \Rightarrow AbA \Rightarrow abA \Rightarrow aba$
- $S \Rightarrow AA \Rightarrow AbA \Rightarrow Aba \Rightarrow aba$
- $S \Rightarrow AA \Rightarrow Aa \Rightarrow aa$
- $S \Rightarrow AA \Rightarrow Aa \Rightarrow bAa \Rightarrow baa$
- $S \Rightarrow AA \Rightarrow Aa \Rightarrow Aba \Rightarrow aba$
- $S \Rightarrow AA \Rightarrow AbA \Rightarrow abA \Rightarrow aba$
- $S \Rightarrow AA \Rightarrow AbA \Rightarrow Aba \Rightarrow aba$
- $S \Rightarrow AA \Rightarrow AAb \Rightarrow aAb \Rightarrow aab$
- $S \Rightarrow AA \Rightarrow AAb \Rightarrow Aab \Rightarrow aab$

Many of these correspond to the same parse trees, just applying the rules in different orders. In any case, the strings that can be generated are: aa, aab, aba, baa.

(b) Notice that  $A \Rightarrow bA \Rightarrow bAb \Rightarrow bab$ , and also that  $A \Rightarrow Ab \Rightarrow bAb \Rightarrow bab$ . This suggests 8 distinct derivations:

- $S \Rightarrow AA \Rightarrow AbA \Rightarrow AbAb \Rightarrow Abab \Rightarrow^* babbab$
- $S \Rightarrow AA \Rightarrow AAb \Rightarrow AbAb \Rightarrow Abab \Rightarrow^* babbab$
- $S \Rightarrow AA \Rightarrow bAA \Rightarrow bAbA \Rightarrow babA \Rightarrow^* babbab$
- $S \Rightarrow AA \Rightarrow AbA \Rightarrow bAbA \Rightarrow babA \Rightarrow^* babbab$

Where each of these four has 2 ways to reach babbab in the last steps. And, of course, one could interleave the productions rather than doing all of the first A, then all of the second A, or vice versa.

(c) This is a matter of formally describing a sequence of applications of the rules in terms of m, n, p that will produce the string  $b^m a b^n a b^p$ .

S

$\Rightarrow^* \text{ by rule } S \rightarrow AA \text{ }^*/$

$$\begin{array}{l}
AA \\
\Rightarrow^* /* \text{ by } m \text{ applications of rule } A \rightarrow bA \text{ */} \\
b^m AA \\
\Rightarrow /* \text{ by rule } A \rightarrow a \text{ */} \\
b^m aA \\
\Rightarrow^* /* \text{ by } n \text{ applications of rule } A \rightarrow bA \text{ */} \\
b^m ab^n A \\
\Rightarrow^* \text{ by } p \text{ applications of rule } A \rightarrow Ab \text{ */} \\
b^m ab^n Ab^p \\
\Rightarrow /* \text{ by rule } A \rightarrow a \text{ */} \\
b^m ab^n ab^p
\end{array}$$

Clearly this derivation (and some variations on it) produce  $b^m ab^n ab^p$  for each  $m, n, p$ .

2. (a)  $G = (V, \Sigma, R, S)$  with  $V = \{S, a, b, c\}$ ,  $\Sigma = \{a, b, c\}$ ,  $R = \{$   
 $S \rightarrow aSa$   
 $S \rightarrow bSb$   
 $S \rightarrow c \quad \}$ .

(b) Same as (a) except remove  $c$  from  $V$  and  $\Sigma$  and replace the last rule,  $S \rightarrow c$ , by  $S \rightarrow \epsilon$ .

(c) This language very similar to the language of (b). (b) was all even length palindromes; this is all palindromes. We can use the same grammar as (b) except that we must add two rules:

$$\begin{array}{l}
S \rightarrow a \\
S \rightarrow b
\end{array}$$

3. This is easy. Recall the inductive definition of regular expressions that was given in class :

1.  $\emptyset$  and each member of  $\Sigma$  is a regular expression.
2. If  $\alpha, \beta$  are regular expressions, then so is  $\alpha\beta$
3. If  $\alpha, \beta$  are regular expressions, then so is  $\alpha\cup\beta$ .
4. If  $\alpha$  is a regular expression, then so is  $\alpha^*$ .
5. If  $\alpha$  is a regular expression, then so is  $(\alpha)$ .
6. Nothing else is a regular expression.

This definition provides the basis for a grammar for regular expressions:

$$\begin{array}{l}
G = (V, \Sigma, R, S) \text{ with } V = \{S, a, b, (, ), \cup, *, \emptyset\}, \Sigma = \{a, b, (, ), \cup, *, \emptyset\}, R = \{ \\
S \rightarrow \emptyset \quad /* \text{ part of rule 1, above} \\
S \rightarrow a \quad /* \quad " \\
S \rightarrow b \quad /* \quad " \\
S \rightarrow SS \quad /* \text{ rule 2} \\
S \rightarrow S \cup S \quad /* \text{ rule 3} \\
S \rightarrow S^* \quad /* \text{ rule 4} \\
S \rightarrow (S) \quad /* \text{ rule 5} \quad \}
\end{array}$$

4. (a) We omit derivations that don't produce strings in  $L$  (i.e, they still contain nonterminals).

$$\begin{array}{l}
L_1 : S \Rightarrow \epsilon \\
L_2 : S \Rightarrow (S) \Rightarrow () \\
L_3 : S \Rightarrow SS \Rightarrow \epsilon S \Rightarrow \epsilon \\
\quad S \Rightarrow (S) \Rightarrow ((S)) \Rightarrow (() \\
L_4 : S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow () \\
\quad S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S) \Rightarrow () \\
\quad S \Rightarrow (S) \Rightarrow ((S)) \Rightarrow (((S))) \Rightarrow (((())) \\
L_5 : S \Rightarrow SS \Rightarrow (S)S \Rightarrow (S)(S) \Rightarrow ()(S) \Rightarrow ()()
\end{array}$$



$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (((S)))S \Rightarrow (((((S))))))$$

$$S \Rightarrow (S) \Rightarrow ((S)) \Rightarrow (((S))) \Rightarrow (((((S)))))) \Rightarrow (((((((S))))))))$$

So  $L_5 = \{\epsilon, (), (()), ((())), (((((S))))), (((((((S))))))\}$

(b) We can give a (weak) upper bound on the number of strings in  $L_K(G)$ . Let  $P$  be the number of rules in  $G$  and let  $N$  be the largest number of nonterminals on the right hand side of any rule in  $G$ . For the first derivation step, we start with  $S$  and have  $P$  choices of derivations to take. So at most  $P$  strings can be generated. (Generally there will be many fewer, since many rules may not apply, but we're only producing an upper bound here, so that's okay.) At the second step, we may have  $P$  strings to begin with (any one of the ones produced in the first step), each of them may have up to  $N$  nonterminals that we could choose to expand, and each nonterminal could potentially be expanded in  $P$  ways. So the number of strings that can be produced is  $P \times N \times P$ . Note that many of them aren't strings in  $L$  since they may still contain nonterminals, but this number is an upper bound on the number of strings in  $L$  that can be produced. At the third derivation step, each of those strings may again have  $N$  nonterminals that can be expanded and  $P$  ways to expand each. In general, an upper bound on the number of strings produced after  $K$  derivation steps is  $P^K N^{(K-1)}$ , which is clearly finite. The key here is that there is a finite number of rules and that each rule produces a string of finite length.

5. We will show that  $L(G)$  is precisely the set of all even length strings in  $\{a, b\}^*$ . But we know that that language is regular. QED.

First we show that only even length strings are generated by  $G$ . This is trivial. Every rule that generates any terminal characters generates two. So only strings of even length can be generated.

Now we must show that all strings of even length are produced. This we do by induction on the length of the strings:

*Base case:*  $\epsilon \in L_G$  (by application of the last rule). So we generate the only string of length 0.

*Induction hypothesis:* All even length strings of length  $\leq N$  (for even  $N$ ) can be generated from  $S$ .

*Induction step:* We need to show that any string of length  $N+2$  can be generated. Any string  $w$  of length  $N+2$  ( $N \geq 0$ ) can be rewritten as  $xyz$ , where  $x$  and  $z$  are single characters and  $|y| = N$ . By the induction hypothesis, we know that all values of  $y$  can be generated from  $S$ . We now consider all possible combinations of values for  $x$  and  $z$  that can be used to create  $w$ . There are four, and the first four rules in the grammar generate, for any string  $T$  derivable from  $S$ , the four strings that contain  $T$  plus a single character appended at the beginning and at the end. Thus all possible strings  $w$  of length  $N+2$  can be generated.

6. Since we already have a grammar for expressions ( $E$ ), we'll just use  $E$  in this grammar and treat it as though it were a terminal symbol. Of course, what we really have to do is to combine this grammar with the one for  $E$ . As we did in our grammar for  $E$ , we'll use the terminal string  $id$  to stand for any identifier.

$G = (V, \Sigma, R, S)$ , where  $V = \{S, U, C, L, T, E, :, =, <, >, ;, a-z, id\}$ ,  $\Sigma = \{ :, =, <, >, ;, a-z, id\}$ , and

$R = \{$

$S \rightarrow L U$	/* a statement can be a label followed by an unlabeled statement
$S \rightarrow U$	/* or a statement can be just an unlabeled statement. We need to make the distinction between $S$ and $U$ if we want to prevent a statement from being preceded by an arbitrary number of labels.
$U \rightarrow id := E$	/* assignment statement
$U \rightarrow if E T E then S$	/* if statement
$U \rightarrow while E T E do S$	/* while statement
$U \rightarrow goto L$	/* goto statement
$U \rightarrow begin S; S end$	/* compound statement
$L \rightarrow id$	/* a label is just an identifier

$T \rightarrow \langle | \rangle =$  /\* we use T to stand for a test operator. We introduce the | (or) notation here for convenience. \*/

There's one problem we haven't addressed here. We have not guaranteed that every label that appears after a goto statement actually appears in the program. In general, this cannot be done with a context-free grammar.

**7. (a)**  $L = \{a^m b^m : m \geq n\}$ . This one is very similar to Example 8 in Supplementary Materials: Context-Free Languages and Pushdown Automata: Designing Context-Free Grammars. The only difference is that in that case,  $m \leq n$ . So you can use any of the ideas presented there to solve this problem.

**(b)**  $L = \{a^m b^n c^p d^q : m + n = p + q\}$ . This one is somewhat like **(a)**: For any string  $a^m b^n c^p d^q \in L$ , we will produce a's and d's in parallel for a while. But then one of two things will happen. Either  $m \geq q$ , in which case we begin producing a's and c's for a while, or  $m \leq q$ , in which case we begin producing b's and d's for a while. (You will see that it is fine that it's ambiguous what happens if  $m = q$ .) Eventually this process will stop and we will begin producing the innermost b's and c's for a while. Notice that any of those four phases could produce zero pairs. Since the four phases are distinct, we will need four nonterminals (since, for example, once we start producing c's, we do not want ever to produce any d's again). So we have:

$$G = (\{S, T, U, V, a, b, c, d\}, \{a, b, c, d\}, R, S), \text{ where}$$

$$R = \{S \rightarrow aSd, S \rightarrow T, S \rightarrow U, T \rightarrow aTc, T \rightarrow V, U \rightarrow bUd, U \rightarrow V, V \rightarrow bVc, V \rightarrow \epsilon\}$$

Every derivation will use symbols S, T, V in sequence or S, U, V in sequence. As a quick check for fencepost errors, note that the shortest string in L is  $\epsilon$ , which is indeed generated by the grammar. (And we do not need any rules  $S \rightarrow \epsilon$  or  $T \rightarrow \epsilon$ .)

How do we know this grammar works? Notice that any string for which  $m = q$  has two distinct derivations:

$$S \Rightarrow^* a^m S d^m \Rightarrow a^m T d^m \Rightarrow a^m V d^m \Rightarrow a^m b^n c^p d^q, \text{ and}$$

$$S \Rightarrow^* a^m S d^m \Rightarrow a^m U d^m \Rightarrow a^m V d^m \Rightarrow a^m b^n c^p d^q$$

Every string  $a^m b^n c^p d^q \in L$  for which  $m \geq q$  has a derivation:

$$\begin{aligned} & S \\ \Rightarrow & /* \text{ by } q \text{ application of rule } S \rightarrow aSd \text{ */} \\ & a^q S d^q \\ \Rightarrow & /* \text{ by rule } S \rightarrow T \text{ */} \\ & a^q T d^q \\ \Rightarrow & /* \text{ by } m - q \text{ application of rule rule } T \rightarrow aTc \text{ */} \\ & a^q a^{m-q} T c^{m-q} d^q = a^m T c^{m-q} d^q \\ \Rightarrow & /* \text{ by rule } T \rightarrow V \text{ */} \\ & a^m V c^{m-q} d^q \\ \Rightarrow & /* \text{ by } n = p - (m - q) \text{ applications of rule } V \rightarrow bVc \text{ */} \\ & a^m b^n V c^{p-(m-q)} c^{m-q} d^q = a^m b^n V c^p d^q \\ \Rightarrow & /* \text{ by rule } V \rightarrow \epsilon \text{ */} \\ & a^m b^n c^p d^q \end{aligned}$$

For the other case ( $m \leq q$ ), you can show the corresponding derivation. So every string in L is generated by G. And it can be shown that no string not in L is generated by G.

**(c)**  $L = \{w \in \{a, b\}^* : w \text{ has twice as many b's as a's}\}$ . This one is sort of tricky. Why? Because L doesn't care about the order in which the a's and b's occur. But grammars do. One solution is:

$$G = (\{S, a, b\}, \{a, b\}, R, S), \text{ where } R = \{S \rightarrow SaSbSbS, S \rightarrow SbSaSbS, S \rightarrow SbSbSaS, S \rightarrow \epsilon\}$$

Try some examples to convince yourself that this grammar works. Why does it work? Notice that all the rules for S preserve the invariant that there are twice as many b's as a's. So we're guaranteed not to generate any strings that aren't in L. Now we just have to worry about generating all the strings that are in L. The first three rules handle the three possible orders in which the symbols b,b, and a can occur.

Another approach you could take is to build a pushdown automaton for  $L$  and then derive a grammar from it. This may be easier simply because PDA's are good at counting. But deriving the grammar isn't trivial either. If you had a hard time with this one, don't worry.

(d)  $L = \{uawb : u, w \in \{a, b\}^*, |u| = |w|\}$ . This one fools some people since you might think that the  $a$  and  $b$  are correlated somehow. But consider the simpler language  $L' = \{uaw : u, w \in \{a, b\}^*, |u| = |w|\}$ . This one seems easier. We just need to generate  $u$  and  $w$  in parallel, keeping something in the middle that will turn into  $a$ . Now back to  $L$ :  $L$  is just  $L'$  with  $b$  tacked on the end. So a grammar for  $L$  is:

$$G = (\{S, T, a, b\}, \{a, b\}, R, S), \text{ where } R = \{S \rightarrow Tb, T \rightarrow aTa, T \rightarrow aTb, T \rightarrow bTa, T \rightarrow bTb, T \rightarrow a\}.$$

8. (a)  $G = (\{S, A, M, D, F, a, b, c\}, \{A, M, D, S, a, b, c\}, R, S)$ , where  $R = \{$

$$\begin{array}{ll} F \rightarrow a & F \rightarrow AFF \\ F \rightarrow b & F \rightarrow SFF \\ F \rightarrow c & F \rightarrow MFF \\ & F \rightarrow DFF \end{array} \}$$

(b) First, we let  $L' = L \cap A^*a^*$ .  $L' = \{A^n a^{n+1} : n \geq 0\}$ .  $L'$  can easily be shown to be nonregular using the Pumping Theorem, so, since the regular languages are closed under intersection,  $L$  must not be regular.

9. (a)  $abbccdd$

(b)  $G = (\{S, X, Y, a, b, c, d\}, \{a, b, c, d\}, R, S)$ , where  $R$  is either:

$$\begin{array}{l} (S \rightarrow aXdd, X \rightarrow Xd, X \rightarrow aXd, X \rightarrow bbYccc, Y \rightarrow bbYccc, Y \rightarrow \epsilon), \text{ or} \\ (S \rightarrow aSd, S \rightarrow Sd, S \rightarrow aMdd, M \rightarrow bbccc, M \rightarrow bbMccc) \end{array}$$

## CS 341 Homework 12

### Parse Trees

1. Consider the grammar  $G = (\{+, *, (, ), \text{id}, T, F, E\}, \{+, *, (, ), \text{id}\}, R, E)$ , where  $R = \{E \rightarrow E+T, E \rightarrow T, T \rightarrow T * F, T \rightarrow F, F \rightarrow (E), F \rightarrow \text{id}\}$ .  
Give two derivations of the string  $\text{id} * \text{id} + \text{id}$ , one of which is leftmost and one of which is not leftmost.
2. Draw parse trees for each of the following:
  - (a) The simple grammar of English we presented in class and the string "big Jim ate green cheese."
  - (b) The grammar of Problem 1 and the strings  $\text{id} + (\text{id} + \text{id}) * \text{id}$  and  $(\text{id} * \text{id} + \text{id} * \text{id})$ .
3. Present a context-free grammar that generates  $\emptyset$ , the empty language.
4. Consider the following grammar (the start symbol is S; the alphabets are implicit in the rules):
$$S \rightarrow SS \mid AAA \mid \epsilon$$
$$A \rightarrow aA \mid Aa \mid b$$
  - (a) Describe the language generated by this grammar.
  - (b) Give a left-most derivation for the terminal string abbaba.
  - (c) Show that the grammar is ambiguous by exhibiting two distinct derivation trees for some terminal string.
  - (d) If this language is regular, give a regular (right linear) grammar generating it and construct the corresponding FSM. If the language is not regular, prove that it is not.
5. Consider the following language :  $L = \{w^R w'' : w \in \{a, b\}^* \text{ and } w'' \text{ indicates } w \text{ with each occurrence of } a \text{ replaced by } b, \text{ and vice versa}\}$ . Give a context-free grammar G that generates L and a parse tree that shows that  $aababb \in L$ .
6. (a) Consider the CFG that you constructed in Homework 11, Problem 2 for  $\{wcw^R : w \in \{a, b\}^*\}$ . How many derivations are there, using that grammar, for the string aabacabaa?  
(b) Show parse tree(s) corresponding to your derivation(s). Is the grammar ambiguous?
7. Consider the language  $L = \{w \in \{a, b\}^* : w \text{ contains equal numbers of } a\text{'s and } b\text{'s}\}$ 
  - (a) Write a context-free grammar G for L.
  - (b) Show two derivations (if possible) for the string aabbab using G. Show at least one leftmost derivation.
  - (c) Do all your derivations result in the same parse tree? If so, see if you can find other parse trees or convince yourself there are none.
  - (d) If G is ambiguous (i.e., you found multiple parse trees), remove the ambiguity. (Hint: look out for two recursive occurrences of the same nonterminal in the right side of a rule, e.g.  $X \rightarrow XX$ )
  - (e) See how many parse trees you get for aabbab using the grammar developed in (d).

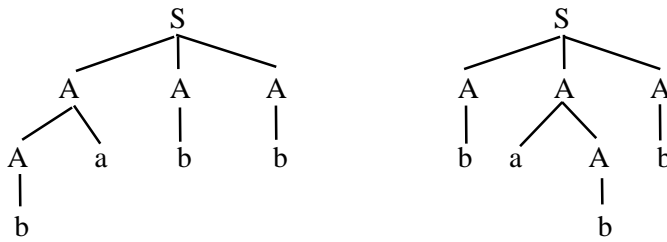
### Solutions

3.  $G = (\{S\} \cup \Sigma, \Sigma, R, S)$ , where R is any set of rules that can't produce any strings in  $\Sigma^*$ . So, for example,  $R = \{S \rightarrow S\}$  does the trick. So does  $R = \emptyset$ .

4. (a)  $(a^*ba^*ba^*ba^*)^*$

(b)  $S \Rightarrow AAA \Rightarrow aAAA \Rightarrow abAA \Rightarrow abAaA \Rightarrow abbaA \Rightarrow abbaAa \Rightarrow abbaba$

(c)



(d)  $G = (\{S, S_1, B_1, B_2, B_3, a, b\}, \{a, b\}, R, S)$ , where  $R = \{$

$S \rightarrow \epsilon$

$B_1 \rightarrow aB_1$

$B_3 \rightarrow aB_3$

$S \rightarrow S_1$

$B_1 \rightarrow bB_2$

$B_3 \rightarrow \epsilon$

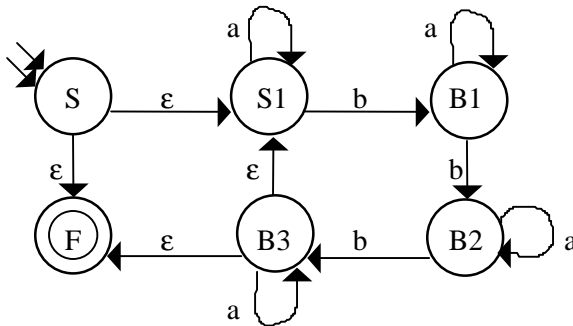
$S_1 \rightarrow aS_1$

$B_2 \rightarrow aB_2$

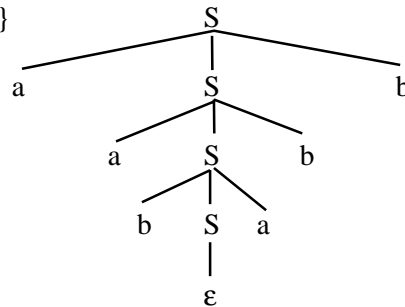
$B_3 \rightarrow S_1$

$S_1 \rightarrow bB_1$

$B_2 \rightarrow bB_3$



5.  $G = (\{S, a, b\}, \{a, b\}, R, S)$ ,  $R = \{ S \rightarrow aSb, S \rightarrow bSa, S \rightarrow \epsilon \}$



6. (a) The grammar is  $G = (V, \Sigma, R, S)$  with  $V = \{S, a, b, c\}$ ,  $\Sigma = \{a, b, c\}$ ,  $R = \{S \rightarrow aSa, S \rightarrow bSb, S \rightarrow c\}$ .

There is a single derivation:

$S \Rightarrow aSA \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabaSabaa \Rightarrow aabacabaa$

(b) There is a single parse tree. The grammar is unambiguous.

7. (a)  $G = (V, \Sigma, R, S)$  with  $V = \{S\}$ ,  $\Sigma = \{a, b\}$ ,  $R = \{$

$S \rightarrow aSb$

$S \rightarrow bSa$

$S \rightarrow \epsilon$

$S \rightarrow SS \}$

(b) (i)  $S \Rightarrow SS \Rightarrow aSbS \Rightarrow aaSbbS \Rightarrow aabbaSb \Rightarrow aabbab$  /\* This is the leftmost derivation of the most "sensible" parse.

(ii)  $S \Rightarrow SS \Rightarrow SSS \Rightarrow aSbSS \Rightarrow aaSbbSS \Rightarrow aabbSS \Rightarrow aabbaSbS \Rightarrow aabbabS \Rightarrow aabbab$  /\* This is the leftmost derivation of a parse that introduced an unnecessary S in the first step, which was then eliminated by rewriting it as  $\epsilon$  in the final step.

(c) No. The two derivations shown here have different parse trees. They do, however, have the same bracketing,  $[a[ab]b][ab]$ . (In other words, they have similar essential structures.) They differ only in how  $S$  is introduced and then eliminated. But there are other derivations that correspond to additional parse trees, and some of them correspond to a completely different bracketing,  $[a[ab][ba]b]$ . One derivation that does this is

$$(ii) S \Rightarrow aSb \Rightarrow aSSb \Rightarrow aabSb \Rightarrow aabbab$$

(d) This is tricky. Recall that we were able to eliminate ambiguity in the case of the balanced parentheses language just by getting rid of  $\epsilon$  except at the very top to allow for the empty string. If we do the same thing here, we get  $R = \{ S \rightarrow \epsilon$

$$\begin{aligned} S &\rightarrow T \\ T &\rightarrow ab \\ T &\rightarrow aTb \\ T &\rightarrow ba \\ T &\rightarrow bTa \\ T &\rightarrow TT \end{aligned}$$

But  $aabbab$  still has multiple parses in this grammar. This language is different from balanced parens since we can go back and forth between being ahead on a's and being ahead on b's (whereas, in the paren language, we must always either be even or be ahead on open paren). So the two parses correspond to the bracketings  $[aabb][ab]$  and  $[a[ab][ba]b]$ . The trouble is the rule  $T \rightarrow TT$ , which can get applied at the very top of the tree (as in the case of the first bracketing shown here), or anywhere further down (as in the case of the second one). We clearly need some capability for forming a string by concatenating a perfectly balanced string with another one, since, without that, we'll get no parse for the string  $abba$ . Just nesting won't work. We have to be able to combine nesting and concatenation, but we have to control it. It's tempting to think that maybe an unambiguous grammar doesn't exist, but it's pretty easy to see how to build a deterministic pda (with a bottom of stack symbol) to accept this language, so there must be an unambiguous grammar. What we need is the notion of an A region, in which we are currently ahead on a's, and a B region, in which we are currently ahead on b's. Then at the top level, we can allow an A region, followed by a B region, followed by an A region and so forth. Think of switching between regions as what will happen when the stack is empty and we're completely even on the number of a's and b's that we've seen so far. For example,  $[ab][ba]$  is one A region followed by one B region. Once we enter an A region, we stay in it, always generating an a followed (possibly after something else embedded in the middle) by a b. After all, the definition of an A region, is that we're always ahead on a's. Only when we are even, can we switch to a B region. Until then, if we want to generate a b, we don't need to do a pair starting with b. We know we're ahead on a's, so make any desired b's go with an a we already have. Once we are even, we must either quit or move to a B region. If we allow for two A regions to be concatenated at the top, there will be ambiguity between concatenating two A regions at the top vs. staying in a single one. We must, however, allow two A regions to be concatenated once we're inside an A region. Consider  $[a[ab][ab]b]$ . Each  $[ab]$  is a perfectly balanced A region and they are concatenated inside the A region whose boundaries are the first a and the final b. So we must distinguish between concatenation within a region (which only happens with regions of the same type, e.g, two A's within an A) and concatenation at the very top level, which only happens between different types.

Also, we must be careful of any rule of the form  $X \rightarrow XX$  for another reason. Suppose we have a string that corresponds to  $XXX$ . Is that the first X being rewritten as two, or the second one being rewritten as two. We need to force a single associativity.

All of this leads to the following set of rules R:

$S \rightarrow \epsilon$   
 $S \rightarrow T_a$  /\* start with an A region, then optionally a B, then an A, and so forth  
 $S \rightarrow T_b$  /\* start with a B region, then optionally an A, then a B, and so forth

$T_a \rightarrow A$  /\* just a single A region  
 $T_a \rightarrow AB$  /\* two regions, an A followed by a B  
 $T_a \rightarrow ABT_a$  /\* we write this instead of  $T_a \rightarrow T_aT_a$  to allow an arbitrary number of regions,  
but force associativity,  
 $T_b \rightarrow B$  /\* these next three rules are the same as the previous three but starting with b  
 $T_b \rightarrow BA$   
 $T_b \rightarrow BAT_b$

$A \rightarrow A_1$  /\* this A region can be composed of a single balanced set of a's and b's  
 $A \rightarrow A_1A$  /\* or it can have arbitrarily many such balanced sets.  
 $A_1 \rightarrow aAb$  /\* a balanced set is a string of A regions inside a matching a, b pair  
 $A_1 \rightarrow ab$  /\* or it bottoms out to just the pair a, b

$B \rightarrow B_1$  /\* these next four rules are the same as the previous four but for B regions  
 $B \rightarrow B_1B$   
 $B_1 \rightarrow bBa$   
 $B_1 \rightarrow ba$

(e) The string aabbab is a single A region, and has only one parse tree in this grammar, corresponding to [[aabb][ab]]. You may also want to try abab, abba, and abaababb to see how G works.

## CS 341 Homework 13 Pushdown Automata

1. Consider the pushdown automaton  $M = (K, \Sigma, \Gamma, \Delta, s, F)$ , where

$$K = \{s, f\},$$

$$F = \{f\},$$

$$\Sigma = \{a, b\},$$

$$\Gamma = \{a\},$$

$$\Delta = \{((s, a, \epsilon), (s, a)), ((s, b, \epsilon), (s, a)), ((s, a, \epsilon), (f, \epsilon)), ((f, a, a), (f, \epsilon)), ((f, b, a), (f, \epsilon))\}.$$

(a) Trace all possible sequences of transitions of  $M$  on input  $aba$ .

(b) Show that  $aba, aa, abb \notin L(M)$ , but  $baa, bab, baaaa \in L(M)$ .

(c) Describe  $L(M)$  in English.

2. Construct pushdown automata that accept each of the following:

(a)  $L$  = the language generated by the grammar  $G = (V, \Sigma, R, S)$ , where

$$V = \{S, (, ), [, ]\},$$

$$\Sigma = \{ (, ), [, ] \},$$

$$R = \{ S \rightarrow \epsilon,$$

$$S \rightarrow SS,$$

$$S \rightarrow [S],$$

$$S \rightarrow (S)\}.$$

(b)  $L = \{a^m b^n : m \leq n \leq 2m\}$ .

(c)  $L = \{w \in \{a, b\}^* : w = w^R\}$ .

(d)  $L = \{w \in \{a, b\}^* : w \text{ has equal numbers of a's and b's}\}$ .

(e)  $L = \{w \in \{a, b\}^* : w \text{ has twice as many a's as b's}\}$ .

(f)  $L = \{a^m b^n : m \geq n\}$

(g)  $L = \{uawb : u \text{ and } w \in \{a, b\}^* \text{ and } |u| = |w|\}$

3. Consider the following language :  $L = \{w^R w : w \in \{a, b\}^* \text{ and } w^R \text{ indicates } w \text{ with each occurrence of } a \text{ replaced by } b, \text{ and vice versa}\}$ . In Homework 12, problem 5, you wrote a context-free grammar for  $L$ . Now give a PDA  $M$  that accepts  $L$  and trace a computation that shows that  $aababb \in L$ .

4. Construct a context-free grammar for the language of problem 2(b):  $L = (\{a^m b^n : m \leq n \leq 2m\})$ .

### Solutions

1. (a) There are three possible computations of  $M$  on  $aba$ :

$$(s, aba, \epsilon) \vdash (s, ba, a) \vdash (s, a, aa) \vdash (s, \epsilon, aaa)$$

$$(s, aba, \epsilon) \vdash (s, ba, a) \vdash (s, a, aa) \vdash (f, \epsilon, aa)$$

$$(s, aba, \epsilon) \vdash (f, ba, \epsilon)$$

None of these is an accepting configuration.

(b) This is done by tracing the computation of  $M$  on each of the strings, as shown in (a).

(c)  $L(M)$  is the set of strings whose middle symbol is  $a$ . In other words,

$$L(M) = \{xay \in \{a, b\}^* : |x| = |y|\}.$$

2. (a) Notice that the square brackets and the parentheses must be properly nested. So the strategy will be to push the open brackets and parens and pop them against matching close brackets and parens as they are read in. We only need one state, since all the counting will be done on the stack. Since  $\epsilon \in L$ , the start state can be final. Thus we have  $M = (\{s\}, \{(, ), [, ]\}, \{(, [, \Delta, s \{s\})\}$ , where (sorry about the confusing use of parentheses both as part of the notation and as symbols in the language):



$$\Delta = \{((s, (, \epsilon), (s, ()), \quad /* \text{push } ( \quad /*$$

$$\quad ((s, [, \epsilon), (s, []), \quad /* \text{push } [ \quad /*$$

$$\quad ((s, ), (, s, \epsilon)), \quad /* \text{if the input character is ) and the top of the stack is (, they match } /*$$

$$\quad ((s, ], ], (s, \epsilon))) \quad /* \text{same for matching square brackets } /*$$

If we run out of input and stack at the same time, we'll accept.

**(b)** Clearly we need to use the stack to count the a's and then compare that count to the b's as they're read in. The complication here is that for every a, there may be either one or two b's. So we'll need nondeterminism. Every string in L has two regions, the a region followed by the b region (okay, they're hard to tell apart in the case of  $\epsilon$ , but trivially, this even true there). So we need a machine with at least two states.

There are two ways we could deal with the fact that, each time we see an a, we don't know whether it will be matched by one b or two. The first is to push either one or two characters onto the stack. In this case, once we get to the b's, we'll pop one character for every b we see. A nondeterministic machine that follows all paths of combinations of one or two pushed characters will find at least one match for every string in L. The alternative is to push a single character for every a and then to get nondeterministic when we're processing the b's: For each stack character, we accept either one b or two. Here's a PDA that takes the second approach. You may want to try writing one that does it the other way. This machine actually needs three states since it needs two states for processing b's to allow for the case where two b's are read but only a single a is popped. So  $M = (\{s, f, g\}, \{a, b\}, \{a\}, \Delta, s, \{f, g\})$ , where

$$\Delta = \{((s, a, \epsilon), (s, a)), \quad /* \text{Read an a and push one onto the stack } /*$$

$$\quad ((s, \epsilon, \epsilon), (f, \epsilon)), \quad /* \text{Jump to the b reading state } /*$$

$$\quad ((f, b, a), (f, \epsilon)), \quad /* \text{Read a single b and pop an a } /*$$

$$\quad ((f, b, a), (g, \epsilon)), /* \text{Read a single b and pop an a but get ready to read a second one } /*$$

$$\quad ((g, b, \epsilon), (f, \epsilon))\}. \quad /* \text{Read a b without popping an a } /*$$

**(c)** A PDA that accepts  $\{w : w = w^R\}$  is just a variation of the PDA that accepts  $\{ww^R\}$  (which you'll find in Lecture Notes 14). You can modify that PDA by adding two transitions  $((s, a, \epsilon), (f, \epsilon))$  and  $((s, b, \epsilon), (f, \epsilon))$ , which have the effect of making odd length palindromes accepted by skipping their middle symbol.

**(d)** We've got another counting problem here, but now order doesn't matter -- just numbers. Notice that with languages of this sort, it's almost always easier to build a PDA than a grammar, since PDAs do a good job of using their stack for counting, while grammars have to consider the order in which characters are generated. If you don't believe this, try writing a grammar for this language.

Consider any string  $w$  in L. At any point in the processing of  $w$ , one of three conditions holds: (1) We have seen equal numbers of a's and b's; (2) We have seen more a's than b's; or (3) We have seen more b's than a's. What we need to do is to use the stack to count whichever character we've seen more of. Then, when we see the corresponding instance of the other character we can "cancel" them by consuming the input and popping off the stack. So if we've seen an excess of a's, there will be a's on the stack. If we've seen an excess of b's there will be b's on the stack. If we're even, the stack will be empty. Then, whatever character comes next, we'll start counting it by putting it on the stack. In fact, we can build our PDA so that the following invariant is always maintained:

$$\begin{aligned} &(\text{Number of a's read so far}) - (\text{Number of b's read so far}) \\ &= \\ &(\text{Number of a's on stack}) - (\text{Number of b's on stack}) \end{aligned}$$

Notice that  $w \in L$  if and only if, when we finish reading  $w$ ,

$$[(\text{Number of a's read so far}) - (\text{Number of b's read so far})] = 0.$$

So, if we build  $M$  so that it maintains this invariant, then we know that if  $M$  consumes  $w$  and ends with its stack empty, it has seen a string in L. And, if its stack isn't empty, then it hasn't seen a string in L.

To make this work, we need to be able to tell if the stack is empty, since that's the only case where we might consider pushing either a or b. Recall that we can't do that just by writing  $\epsilon$  as the stack character, since that always matches, even if the stack is not empty. So we'll start by pushing a special character # onto the bottom of the stack. We can then check to see if the stack is empty by seeing if # is on top. We can do all the real work in our PDA in a single state. But, because we're using the bottom of stack symbol #, we need two additional states: the start state, in which we do nothing except push # and move to the working state, and the final state, which we get to once we've popped # and can then do nothing else. Considering all these issues, we get  $M = (\{s, q, f\}, \{a, b\}, \{\#, a, b\}, \Delta, s, \{f\})$ , where

$$\Delta = \{ ((s, \epsilon, \epsilon), (q, \#)), \quad /* \text{ push } \# \text{ and move to the working state } q \text{ */}$$

$$((q, a, \#), (q, a\#)), \quad /* \text{ the stack is empty and we've got an } a, \text{ so push it } */$$

$$((q, a, a), (q, aa)), \quad /* \text{ the stack is counting } a\text{'s and we've got another one so push it } */$$

$$((q, b, a), (q, \epsilon)), \quad /* \text{ the stack is counting } a\text{'s and we've got } b, \text{ so cancel } a \text{ and } b \text{ */}$$

$$((q, b, \#), (q, b\#)), \quad /* \text{ the stack is empty and we've got a } b, \text{ so push it } */$$

$$((q, b, b), (q, bb)), \quad /* \text{ the stack is counting } b\text{'s and we've got another one so push it } */$$

$$((q, a, b), (q, \epsilon)), \quad /* \text{ the stack is counting } b\text{'s and we've got } a, \text{ so cancel } b \text{ and } a \text{ */}$$

$$((q, \epsilon, \#), (f, \epsilon)) \}. \quad /* \text{ the stack is empty of } a\text{'s and } b\text{'s. Pop the } \# \text{ and quit. } */$$

To convince yourself that  $M$  does the job, you should show that  $M$  does in fact maintain the invariant we stated above.

The only nondeterminism in this machine involves the last transition in which we guess that we're at the end of the input. There is an alternative way to solve this problem in which we don't bother with the bottom of stack symbol #. Instead, we substitute a lot of nondeterminism and we sometimes push a's on top of b's, and so forth. Most of those paths will end up in dead ends. The machine has fewer states but is harder to analyze. Try to construct it if you like.

(e) This one is similar to (d) except that there are two a's for every b. Recall the two techniques for matching two to one that we discussed in our solution to (b). This time, though, we do know that there are always two a's to every b. We don't need nondeterminism to allow for either one *or* two. But, because we no longer know that all the a's come first, we do need to consider what to do in the two cases: (1) We're counting b's on the stack; and (2) We're counting a's on the stack. If we're counting b's, let's take the approach in which we push two b's every time we see one. Then, when we go to cancel a's, we can just pop one b for each a. If we see twice as many a's as b's, we'll end up with an empty stack. Now what if we're counting a's? We'll push one a for every one we see. When we see b, we pop two a's. The only special case we need to consider arises in strings such as "aba", where we'll only have seen a single a at the point at which we see the b. What we need to do is to switch from counting a's to counting b's, since the b counts twice. Thus the invariant that we want to maintain is now

$$\begin{aligned} &(\text{Number of } a\text{'s read so far}) - 2 * (\text{Number of } b\text{'s read so far}) \\ &= \\ &(\text{Number of } a\text{'s on stack}) - (\text{Number of } b\text{'s on stack}) \end{aligned}$$

We can do all this with  $M = (\{s, q, f\}, \{a, b\}, \{\#, a, b\}, \Delta, s, \{f\})$ , where

$$\Delta = \{ ((s, \epsilon, \epsilon), (q, \#)), \quad /* \text{ push } \# \text{ and move to the working state } q \text{ */}$$

$$((q, a, \#), (q, a\#)), \quad /* \text{ the stack is empty and we've got an } a, \text{ so push it } */$$

$$((q, a, a), (q, aa)), \quad /* \text{ the stack is counting } a\text{'s and we've got another one so push it } */$$

$$((q, b, aa), (q, \epsilon)), \quad /* \text{ the stack is counting } a\text{'s and we've got } b, \text{ so cancel } aa \text{ and } b \text{ */}$$

$$((q, b, a\#), (q, b\#)), \quad /* \text{ the stack contains a single } a \text{ and we've got } b, \text{ so cancel the } a \text{ and } b$$

$$\quad \text{and start counting } b\text{'s, since we have a shortage of one } a \text{ */}$$

$$((q, b, \#), (q, bb\#)), \quad /* \text{ the stack is empty and we've got a } b, \text{ so push two } b\text{'s } */$$

$$((q, b, b), (q, bbb)), \quad /* \text{ the stack is counting } b\text{'s and we've got another one so push two } */$$

$$((q, a, b), (q, \epsilon)), \quad /* \text{ the stack is counting } b\text{'s and we've got } a, \text{ so cancel } b \text{ and } a \text{ */}$$

$$((q, \epsilon, \#), (f, \epsilon)) \}. \quad /* \text{ the stack is empty of } a\text{'s and } b\text{'s. Pop the } \# \text{ and quit. } */$$

You should show that  $M$  preserves the invariant above.

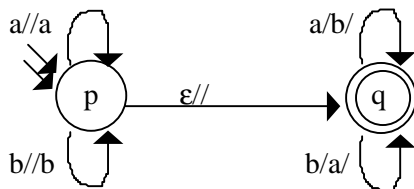
(f) The idea here is to push each  $a$  as we see it. Then, on the first  $b$ , move to a second state and pop an  $a$  for each  $b$ . If we get to the end of the string and either the stack is empty ( $m = n$ ) or there are still  $a$ 's on the stack ( $m > n$ ) then we accept. If we find a  $b$  and there's no  $a$  to pop, then there will be no action and so we'll fail. This machine is nondeterministic for two reasons. The first is that, in case there are no  $b$ 's, we must be able to guess that we've reached the end of the input string and go pop all the  $a$ 's off the stack. The second is that if there were  $b$ 's but fewer than the number of  $a$ 's, then we must guess that we've reached the end of the input string and pop all the  $a$ 's off the stack. If we guess wrong, that path will just fail, but it will never cause us to accept something we shouldn't, since it only pops off extra  $a$ 's, which is what we want anyway. We don't need a separate state for the final popping phase, since we're willing to accept either  $m = n$  or  $m > n$ . This contrasts with the example we did in class, where we required that  $m > n$ . In that case, the state where we run out of input and the stack at the same time (i.e.,  $m = n$ ) had to be a rejecting state. Thus we needed an additional accepting state where we popped the stack.

$M = (\{1, 2\}, \{a, b\}, \{a\}, 1, \{2\}, \Delta =$   
 $((1, a, \epsilon), (1, a))$  /\* push an  $a$  on the stack for every input  $a$   
 $((1, b, a), (2, \epsilon))$  /\* pop an  $a$  for the first  $b$  and go to the  $b$ -popping state  
 $((1, \epsilon, \epsilon), (2, \epsilon))$  /\* in case there aren't any  $b$ 's -- guess end of string and go pop any  $a$ 's  
 $((2, b, a), (2, \epsilon))$  /\* for each input  $b$ , pop an  $a$  off the stack  
 $((2, \epsilon, a), (2, \epsilon))$  /\* if we run out of input while there are still  $a$ 's on the stack,  
then pop the  $a$ 's and accept

(g) The idea here is to create a nondeterministic machine. In the start state (1), it reads  $a$ 's and  $b$ 's, and for each character seen, it pushes an  $x$  on the stack. Thus it counts the length of  $u$ . If it sees an  $a$ , it may also guess that this is the required separator  $a$  and go to state 2. In state 2, it reads  $a$ 's and  $b$ 's, and for each character seen, pops an  $x$  off the stack. If there's nothing to pop, the machine will fail. If it sees a  $b$ , it may also guess that this is the required final  $b$  and go to the final state, state 3. The machine will then accept if both the input and the stack are empty.

$M = (\{1, 2, 3\}, \{a, b\}, \{x\}, s, \{2\}, \Delta =$   
 $((1, a, \epsilon), (1, x))$  /\* push an  $x$  on the stack for every input  $a$   
 $((1, b, \epsilon), (1, x))$  /\* push an  $x$  on the stack for every input  $b$   
 $((1, a, \epsilon), (2, \epsilon))$  /\* guess that this is the separator  $a$ . No stack action  
 $((2, a, x), (2, \epsilon))$  /\* for each input  $a$ , pop an  $x$  off the stack  
 $((2, b, x), (2, \epsilon))$  /\* for each input  $b$ , pop an  $x$  off the stack  
 $((2, b, \epsilon), (3, \epsilon))$  /\* guess that this is the final  $b$  and go to the final state

3.



4.  $S \rightarrow \epsilon$   
 $S \rightarrow aSb$   
 $S \rightarrow aSbb$

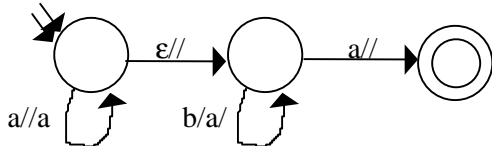
## CS 341 Homework 14 Pushdown Automata and Context-Free Grammars

1. In class, we described an algorithm for constructing a PDA to accept a language  $L$ , given a context free grammar for  $L$ . Let  $L$  be the balanced brackets language defined by the grammar  $G = (\{S, [, ]\}, \{[, ]\}, R, S)$ , where  $R =$

$$S \rightarrow \epsilon, S \rightarrow SS, S \rightarrow [S]$$

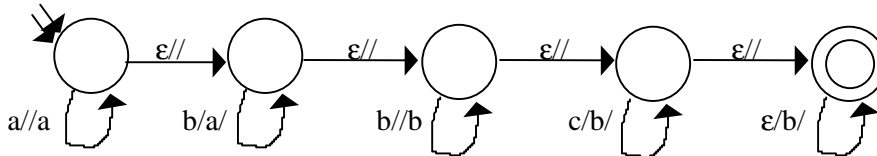
Apply the construction algorithm to this grammar to derive a PDA that accepts  $L$ . Trace the operation of the PDA you have constructed on the input string  $[[[]]]$ .

2. Consider the following PDA  $M$ :



- (a) What is  $L(M)$ ?
- (b) Give a deterministic PDA that accepts  $L(M)$  (*not*  $L(M)\$$ ).

3. Write a context-free grammar for  $L(M)$ , where  $M$  is



- 4. Consider the language  $L = \{ba^{m_1}ba^{m_2}b\dots ba^{m_n} : n \geq 2, m_1, \dots, m_n \geq 0, \text{ and } m_i \neq m_j \text{ for some } i, j\}$ 
  - (a) Give a nondeterministic PDA that accepts  $L$ .
  - (b) Write a context-free grammar that generates  $L$ .
  - (c) Prove that  $L$  is not regular.

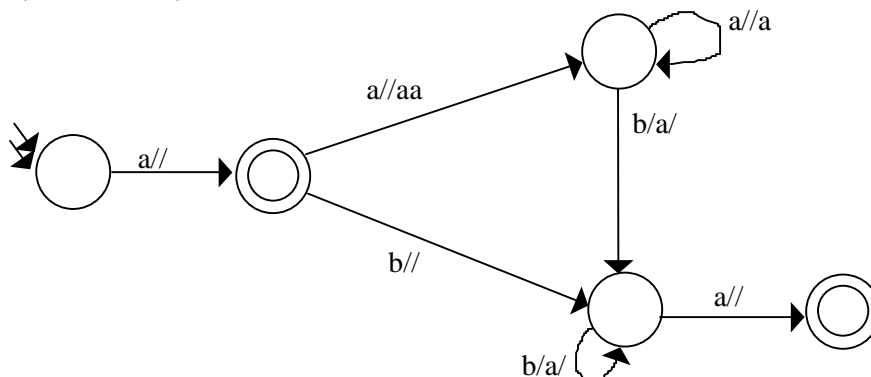
### Solutions

1. This is a very simple mechanical process that you should have no difficulty carrying out, and getting the following PDA,  $M = (\{p, q\}, \{[, ]\}, \{S, [, ]\}, \Delta, p, \{q\})$ , where

$$\Delta = \{((p, \epsilon, \epsilon), (q, S)), ((q, \epsilon, S), (q, \epsilon)), ((q, \epsilon, S), (q, SS)), ((q, \epsilon, S), (q, [S])), ((q, [, ], (q, \epsilon)), ((q, ], ], (q, \epsilon)))\}$$

2. (a)  $L(M) = \{a^n b^n a : n \geq 0\}$

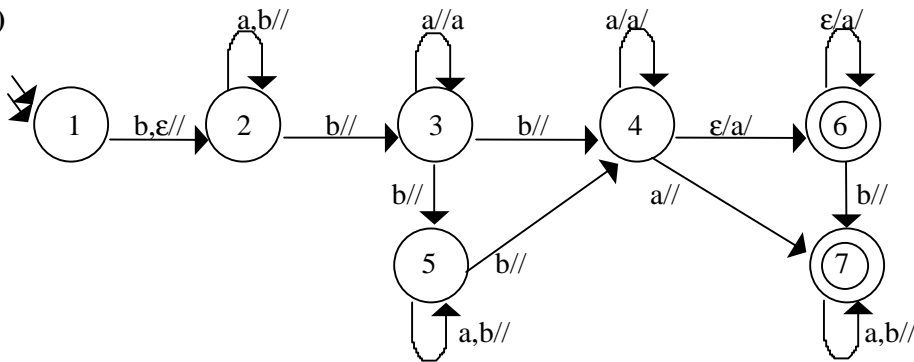
(b)



3. Don't even try to use the grammar construction algorithm. Just observe that  $L = \{a^n b^m c^p : m \geq p \text{ and } n \text{ and } p \geq 0\}$ , or, alternatively  $\{a^n b^m c^p : m \geq n + p \text{ and } n \text{ and } p \geq 0\}$ . It can be generated by the following rules:

$S \rightarrow S_1 S_2$   
 $S_1 \rightarrow a S_1 b$  /\*  $S_1$  generates the  $a^n b^n$  part. \*/  
 $S_1 \rightarrow \epsilon$   
 $S_2 \rightarrow b S_2$  /\*  $S_2$  generates the  $b^m c^p$  part. \*/  
 $S_2 \rightarrow b S_2 c$   
 $S_2 \rightarrow \epsilon$

4. (a)



We use state 2 to skip over an arbitrary number of  $ba^i$  groups that aren't involved in the required mismatch.

We use state 3 to count the first group of a's we care about.

We use state 4 to count the second group and make sure it's not equal to the first.

We use state 5 to skip over an arbitrary number of  $ba^i$  groups in between the two we care about.

We use state 6 to clear the stack in the case that the second group had fewer a's than the first group did.

We use state 7 to skip over any remaining  $ba^i$  groups that aren't involved in the required mismatch.

(b)  $S \rightarrow A' b L A'$  /\* L will take care of two groups where the first group has more a's \*/  
 $S \rightarrow A' b R A'$  /\* R will take care of two groups where the second group has more a's \*/  
 $L \rightarrow ab \mid aL \mid aLa$   
 $R \rightarrow ba \mid Ra \mid aRa$   
 $A' \rightarrow b A A' \mid \epsilon$   
 $A \rightarrow a A \mid \epsilon$

(c) Let  $L_1 = ba^*ba^*$ , which is obviously regular.

If L is regular then

$L_2 = L \cap L_1$  is regular.

$L_2 = ba^n ba^m, n \neq m$

$\neg L_2 \cap L_1$  must also be regular.

But  $\neg L_2 \cap L_1 = ba^n ba^m, n = m$ , which can easily be shown, using the pumping theorem, not to be regular.

## CS 341 Homework 15

### Parsing

1. Show that the following languages are deterministic context free.

- (a)  $\{a^m b^n : m \neq n\}$
- (b)  $\{wcw^R : w \in \{a, b\}^*\}$
- (c)  $\{ca^m b^m : m \geq 0\} \cup \{da^m b^{2m} : m \geq 0\}$
- (d)  $\{a^m cb^m : m \geq 0\} \cup \{a^m db^{2m} : m \geq 0\}$

2. Consider the context-free grammar:  $G = (V, \Sigma, R, S)$ , where  $V = \{(, ), \cdot, a, S, A\}$ ,  $\Sigma = \{(, ), \cdot\}$ , and  $R =$

- $\{ S \rightarrow (),$
- $S \rightarrow a,$
- $S \rightarrow (A),$
- $A \rightarrow S,$
- $A \rightarrow A.S\}$

(If you are familiar with the programming language LISP, notice that  $L(G)$  contains all atoms and lists, where the symbol  $a$  stands for any non-null atom.)

(a) Apply left factoring and the rule for getting rid of left recursion to  $G$ . Let  $G'$  be the resulting grammar. Argue that  $G'$  is LL(1). Construct a deterministic pushdown automaton  $M$  that accepts  $L(G)\$$  by doing a top down parse. Study the computation of  $M$  on the string  $((()).a)$ .

(b) Repeat Part (a) for the grammar resulting from  $G$  if one replaces the first rule by  $A \rightarrow \epsilon$ .

(c) Repeat Part (a) for the grammar resulting from  $G$  if one replaces the last rule by  $A \rightarrow S.A$ .

3. Answer each of the following questions True or False. If you choose false, you should be able to state a counterexample.

- (a) If a language  $L$  can be described by a regular expression, we can be sure it is a context-free language.
- (b) If a language  $L$  cannot be described by a regular expression, we can be sure it is not a context-free language.
- (c) If  $L$  is generated by a context-free grammar, then  $L$  cannot be regular.
- (d) If there is no pushdown automaton accepting  $L$ , then  $L$  cannot be regular.
- (e) If  $L$  is accepted by a nondeterministic finite automaton, then there is some deterministic PDA accepting  $L$ .
- (f) If  $L$  is accepted by a deterministic PDA, then  $L'$  (the complement of  $L$ ) must be regular.
- (g) If  $L$  is accepted by a deterministic PDA, then  $L'$  must be context free.
- (h) If, for a given  $L$  in  $\{a, b\}^*$ , there exist  $x, y, z$ , such that  $y \neq \epsilon$  and  $xy^n z \in L$  for all  $n \geq 0$ , then  $L$  must be regular.
- (i) If, for a given  $L$  in  $\{a, b\}^*$ , there do not exist  $u, v, x, y, z$  such that  $|vy| \geq 1$  and  $uv^n xy^n z \in L$  for all  $n \geq 0$ , then  $L$  cannot be regular.
- (j) If  $L$  is regular and  $L = L1 \cap L2$  for some  $L1$  and  $L2$ , then at least one of  $L1$  and  $L2$  must be regular.
- (k) If  $L$  is context free and  $L = L1L2$  for some  $L1$  and  $L2$ , then  $L1$  and  $L2$  must both be context free.
- (l) If  $L$  is context free, then  $L^*$  must be regular.
- (m) If  $L$  is an infinite context-free language, then in any context-free grammar generating  $L$  there exists at least one recursive rule.
- (n) If  $L$  is an infinite context-free language, then there is some context-free grammar generating  $L$  that has no rule of the form  $A \rightarrow B$ , where  $A$  and  $B$  are nonterminal symbols.
- (o) Every context-free grammar can be converted into an equivalent regular grammar.
- (p) Given a context-free grammar generating  $L$ , every string in  $L$  has a right-most derivation.

4. Recall problem 4 from Homework 12. It asked you to consider the following grammar for a language  $L$  (the start symbol is  $S$ ; the alphabets are implicit in the rules):

- $S \rightarrow SS \mid AAA \mid \epsilon$
- $A \rightarrow aA \mid Aa \mid b$

(a) It is not possible to convert this grammar to an equivalent one in Chomsky Normal Form. Why not?

(b) Modify the grammar as little as possible so that it generates  $L - \epsilon$ . Now convert this new grammar to Chomsky Normal Form. Is the resulting grammar still ambiguous? Why or why not?

(c) From either the original grammar for  $L - \epsilon$  or the one in Chomsky Normal Form, construct a PDA that accepts  $L - \epsilon$ .

5. Consider the following language :  $L = \{w^R w^n : w \in \{a, b\}^* \text{ and } w^n \text{ indicates } w \text{ with each occurrence of } a \text{ replaced by } b, \text{ and vice versa}\}$ . In Homework 12, problem 5, you wrote a context-free grammar for  $L$ . Then, in Homework 13, problem 3, you wrote a PDA  $M$  that accepts  $L$  and traced one of its computations. Now decide whether you think  $L$  is deterministic context free. Defend your answer.

6. Convert the following grammar for arithmetic expressions to Chomsky Normal Form:

- $E \rightarrow E + T$
- $E \rightarrow T$
- $T \rightarrow T * F$
- $T \rightarrow F$
- $F \rightarrow (E)$
- $F \rightarrow \text{id}$

7. Again, consider the grammar for arithmetic expressions given in Problem 6. Walk through the process of doing a top down parse of the following strings using that grammar. Point out the places where a decision has to be made about what to do.

(a)  $\text{id} * \text{id} + \text{id}$

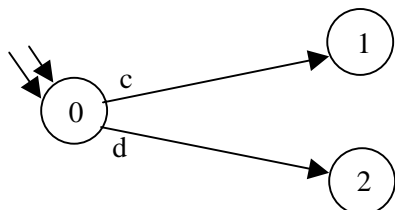
(b)  $\text{id} * \text{id} * \text{id}$

### Solutions

1. (a)  $L = \{a^m b^n : m \neq n\}$ . To show that a language  $L$  is deterministic context free, we need to show a deterministic PDA that accepts  $L\$$ . We did that for  $L = \{a^m b^n : m \neq n\}$  in class. (See Lecture Notes 14).

(b)  $L = \{wcw^R : w \in \{a, b\}^*\}$ . In class (again see Lecture Notes 14), we built a deterministic PDA to accept  $L = \{wcw^R : w \in \{a, b\}^*\}$ . It's easy to turn it into a deterministic PDA that accepts  $L\$$ .

(c)  $L = \{ca^m b^m : m \geq 0\} \cup \{da^m b^{2m} : m \geq 0\}$ . Often it's hard to build a deterministic PDA for a language that is formed by taking the union of two other languages. For example,  $\{a^m b^m : m \geq 0\} \cup \{a^m b^{2m} : m \geq 0\}$  would be hard (in fact it's impossible) because we have no way of knowing, until we run out of  $b$ 's, whether we're expecting two  $b$ 's for each  $a$  or just one. However,  $\{ca^m b^m : m \geq 0\} \cup \{da^m b^{2m} : m \geq 0\}$  is actually quite easy. Every string starts with a  $c$  or a  $d$ . If it's a  $c$ , then we know to look for one  $b$  for each  $a$ ; if it's a  $d$ , then we know to look for two. So the first thing we do is to start our machine like this:



The machine that starts in state 1 is our classic machine for  $a^n b^n$ , except of course that it must have a final transition on  $\$$  to its final state.

We have two choices for the machine that starts in state 2. It can either push one  $a$  for every  $a$  it sees, and then pop an  $a$  for every pair of  $b$ 's, or it can push two  $a$ 's for every  $a$  it sees, and then pop one  $a$  for every  $b$ .

(d)  $L = \{a^m c b^m : m \geq 0\} \cup \{a^m d b^{2m} : m \geq 0\}$ . Now we've got another unioned language. But this time we don't get a clue from the first character which part of the language we're dealing with. That turns out to be okay though, because we do find out before we have to start processing the b's whether we've got two b's for each a or just one. Recall the two approaches we mentioned for machine 2 in the last problem. What we need here is the first, the one that pushes a single a for each a it sees. Then, when we see a c or d, we branch and either pop an a for each b or pop an a for every two b's.

2. (a) We need to apply left factoring to the two rules  $S \rightarrow ()$  and  $S \rightarrow (A)$ . We also need to eliminate the left recursion from  $A \rightarrow A.S$ . Applying left factoring, we get the first column shown here. Then getting rid of left recursion gets us the second column:

$S \rightarrow (S'$	$S \rightarrow (S'$
$S' \rightarrow )$	$S' \rightarrow )$
$S' \rightarrow A)$	$S' \rightarrow A)$
$S \rightarrow a$	$S \rightarrow a$
$A \rightarrow S$	$A \rightarrow SA'$
$A \rightarrow A.S$	$A' \rightarrow .SA'$
	$A' \rightarrow \epsilon$

(b) Notice that the point of the first rule, which was  $S \rightarrow ()$ , was to get a set of parentheses with no A inside. An alternative way to do that is to dump that rule but to add the rule  $A \rightarrow \epsilon$ . Now we always introduce an A when we expand S, but we can get rid of it later. If we do this, then there's no left factoring to be done. We still have to get rid of the left recursion on A, just as we did above, however.

(c) If we change  $A \rightarrow A.S$  to  $S \rightarrow S.A$ , then there's no left recursion to get rid of and we can leave the rules unchanged. Notice, though, that we'll get different parse trees this way, which may or may not be important. To see this, consider the string (a.a.a) and parse it using both the original grammar and the one we get if we change the last rule.

3. (a) True, since all regular languages are context-free.

(b) False, there exist languages that are context-free but not regular.

(c) False. All regular languages are also context-free and thus are generated by context-free grammars.

(d) True, since if L were regular, it would also be context free and thus would be accepted by some PDA.

(e) True, since there must also be a deterministic FSM and thus a deterministic PDA.

(f) False. Consider  $L = a^n b^n$ .  $L' = \{w \in \{a, b\}^* : \text{either some } b \text{ comes before some } a \text{ or there is an unequal number of } a\text{'s and } b\text{'s}\}$ . Clearly this language is not regular since we can't count the a's and b's.

(g) True, since the deterministic context-free languages are closed under complement.

(h) False. Suppose  $L = a^n c^* b^n$ , which is clearly not regular. Let  $x = aa$ ,  $y = c$ , and  $z = bb$ .  $xy^n z \in L$ .

(i) False. L could be finite.

(j) False.  $L_1$  could be  $a^n b^n$  and  $L_2$  could be  $\{\epsilon \cup a^n b^m : n \neq m\}$ . Neither is regular. But  $L_1 \cap L_2 = \{\epsilon\}$ , which is regular.

(k) False. Let  $L_1 = a^*$  and  $L_2 = \{a^n b^m c^m : n \neq m\}$ .  $L_2$  is not context free. But  $L = L_1 L_2 = a^* b^m c^m$ , which is context free.

(l) False. Let  $L = ww^R$ .

(m) True.

(n) True, since we have a procedure for eliminating such unit productions.

(o) False, since there exist context-free languages that are not regular.

(p) True.

4. (a) No grammar in Chomsky Normal Form can generate  $\epsilon$ , yet  $\epsilon \in L$ .



(b) In the original grammar, we could generate zero copies of AAA (by letting S go to  $\epsilon$ ), one copy of AAA (by letting S go to AAA), two copies (by letting S go to SS and then each of them to AAA), three copies of AAA (by letting S go to SS, then one of the S's goes to SS, then all three go to AAA), and so forth. We want to make sure that we can still get one copy, two copies, three copies, etc. We just want to eliminate the zero copies option. Note that the only role of S is to determine how many copies of AAA are produced. Once we have generated A's we can never go back and apply the S rules again. So all we have to do is to eliminate the production  $S \rightarrow \epsilon$ . The modified grammar to accept  $L - \epsilon$  is thus:

$$G = (\{S, A, B, C, a, b\}, \{a, b\}, R, S), \text{ where } R = \{ \\ S \rightarrow SS \mid AAA \\ A \rightarrow aA \mid Aa \mid b$$

If we convert this grammar to Chomsky Normal Form, we get:

$$G = (\{S, A, B, C, a, b\}, \{a, b\}, R, S), \text{ where } R = \{ \\ S \rightarrow SS \quad A \rightarrow AC \\ S \rightarrow AB \quad A \rightarrow b \\ B \rightarrow AA \quad C \rightarrow a \\ A \rightarrow CA \}$$

This grammar is still ambiguous.

(c) (from the grammar of part (b)):  $M = (\{p, q\}, \{a, b\}, \{S, A, a, b\}, \Delta, p, \{q\})$

$$\Delta = \{ \begin{array}{ll} ((p, \epsilon, \epsilon), (q, S)) & ((q, \epsilon, A), (q, aA)) \\ ((q, \epsilon, S), (q, SS)) & ((q, \epsilon, A), (q, Aa)) \\ ((q, \epsilon, S), (q, AAA)) & ((q, \epsilon, A), (q, b)) \\ & ((q, a, a), (q, \epsilon)) \\ & ((q, b, b), (q, \epsilon)) \end{array} \}$$

5. L is not deterministic context free for essentially the same reason that  $ww^R$  is not.

6. The original grammar was:

$$\begin{array}{l} E \rightarrow E + T \\ E \rightarrow T \\ T \rightarrow T * F \\ T \rightarrow F \\ F \rightarrow (E) \\ F \rightarrow id \end{array}$$

Step 2. There are no  $\epsilon$  rules. We show steps 3, 4, and 5 next to each other, so it's clear where the rules in steps 4 and 5 came from. In each case, the first rule that is derived from a step 3 rule is next to its source. If more than one rule is derived from any given step 3 rule, the second and others are shown immediately under the first. That's why there are some blank lines in the first two columns.

Step 3.

$E \Rightarrow^* T, F$   
 $T \Rightarrow^* F$

$G'' = E \rightarrow E + T$

$T \rightarrow T * F$

$F \rightarrow (E)$

$F \rightarrow id$

Then we add:

$E \rightarrow T * F$

$E \rightarrow (E)$

$E \rightarrow id$

$T \rightarrow (E)$

$T \rightarrow id$

Step 4.

$E \rightarrow EPT$

$T \rightarrow TMF$

$F \rightarrow LER$

$F \rightarrow id$

$E \rightarrow TMF$

$E \rightarrow LER$

$E \rightarrow id$

$T \rightarrow LER$

$T \rightarrow id$

$P \rightarrow +$

$M \rightarrow *$

$L \rightarrow ($

$R \rightarrow )$

Step 5.

$E \rightarrow E E'$

$E' \rightarrow PT$

$T \rightarrow T T'$

$T' \rightarrow M F$

$F \rightarrow L F'$

$F' \rightarrow E R$

$F \rightarrow id$

$E \rightarrow T T'$

(since  $T' \rightarrow M F$ )

$E \rightarrow L F'$

(since  $F' \rightarrow E R$ )

$E \rightarrow id$

$T \rightarrow L F'$

(since  $F' \rightarrow E R$ )

$T \rightarrow id$

$P \rightarrow +$

$M \rightarrow *$

$L \rightarrow ($

$R \rightarrow )$

## CS 341 Homework 16

### Languages that Are and Are Not Context-Free

1. Show that the following languages are context-free. You can do this by writing a context free grammar or a PDA, or you can use the closure theorems for context-free languages. For example, you could show that  $L$  is the union of two simpler context-free languages.

- (a)  $L = a^n c b^n$
- (b)  $L = \{a, b\}^* - \{a^n b^n : n \geq 0\}$
- (c)  $L = \{a^m b^n c^p d^q : n = q, \text{ or } m \leq p \text{ or } m + n = p + q\}$
- (d)  $L = \{a, b\}^* - L_1$ , where  $L_1$  is the language  $\{babaabaaab\dots ba^{n-1}ba^n b : n \geq 1\}$ .

2. Show that the following languages are not context-free.

- (a)  $L = \{a^{n^2} : n \geq 0\}$
- (b)  $L = \{www : w \in \{a, b\}^*\}$
- (c)  $L = \{w \in \{a, b, c\}^* : w \text{ has equal numbers of } a\text{'s, } b\text{'s, and } c\text{'s}\}$
- (d)  $L = \{a^n b^m a^n : n \geq m\}$
- (e)  $L = \{a^n b^m c^n d^{(n+m)} : m, n \geq 0\}$

3. Give an example of a context free language ( $\neq \Sigma^*$ ) that contains a subset that is not context free. Describe the subset.

4. What is wrong with the following "proof" that  $a^n b^{2n} a^n$  is context free?

- (1) Both  $\{a^n b^n : n \geq 0\}$  and  $\{b^n a^n : n \geq 0\}$  are context free.
- (2)  $a^n b^{2n} a^n = \{a^n b^n\} \{b^n a^n\}$
- (3) Since the context free languages are closed under concatenation,  $a^n b^{2n} a^n$  is context free.

5. Consider the following context free grammar:  $G = (\{S, A, a, b\}, \{a, b\}, R, S)$ , where  $R = \{$

- $S \rightarrow aAS$
- $S \rightarrow a$
- $A \rightarrow SbA$
- $A \rightarrow SS$
- $A \rightarrow ba \quad \}$

(a) Answer each of the following questions True or False:

- (i) From the fact that  $G$  is context free, it follows that there is no regular expression for  $L(G)$ .
- (ii)  $L(G)$  contains no strings of length 3.
- (iii) For any string  $w \in L(G)$ , there exists  $u, v, x, y, z$  such that  $w = uvxyz$ ,  $|vy| \geq 1$ , and  $uv^n xy^n z \in L(G)$  for all  $n \geq 0$ .
- (iv) If there exist languages  $L_1$  and  $L_2$  such that  $L(G) = L_1 \cup L_2$ , then  $L_1$  and  $L_2$  must both be context free.
- (v) The language  $(L(G))^R$  is context free.

- (b) Give a leftmost derivation according to  $G$  of  $aaaabaa$ .
- (c) Give the parse tree corresponding to the derivation in (b).
- (d) Give a nondeterministic PDA that accepts  $L(G)$ .

6. Show that the following language is context free:

$$L = \{xx^R yy^R zz^R : x, y, z \in \{a, b\}^*\}.$$

7. Suppose that  $L$  is context free and  $R$  is regular.

(a) Is  $L - R$  necessarily context free?

(b) Is  $R - L$  necessarily context free?

8. Let  $L_1 = \{a^n b^m : n \geq m\}$ . Let  $R_1 = \{(a \cup b)^* : \text{there is an odd number of a's and an even number of b's}\}$ . Show a pda that accepts  $L_1 \cap R_1$ .

## Solutions

1. (a)  $L = a^n c b^n$ . We can easily do this one by building a CFG for  $L$ . Our CFG is almost the same as the one we did in class for  $a^n b^n$ :

$$S \rightarrow aSB$$

$$S \rightarrow c$$

(b)  $L = \{a, b\}^* - \{a^n b^n : n \geq 0\}$ . In other words, we've got the complement of  $a^n b^n$ . So we look at how a string could fail to be in  $a^n b^n$ . There are two ways: either the a's and b's are out of order or there are not equal numbers of them. So our language  $L$  is the union of two other languages:

- $L_1 = (a \cup b)^* - a^* b^*$  (strings where the a's and b's are out of order)

- $L_2 = a^n b^m \ n \neq m$  (strings where the a's and b's are in order but there aren't matching numbers of them)

$L_1$  is context free. We gave a context-free grammar for it in class (Lecture 12).  $L_2$  is the complement of the regular language  $a^* b^*$ , so it is also regular and thus context free. Since the union of two context-free languages is context free,  $L$  is context free.

(c)  $L = \{a^m b^n c^p d^q : n = q \text{ or } m \leq p \text{ or } m + n = p + q\}$ . This one looks scary, but it's just the union of three quite simple context-free languages:

$$L_1 = a^m b^n c^p d^q : n = q$$

$$L_2 = a^m b^n c^p d^q : m \leq p$$

$$L_3 = a^m b^n c^p d^q : m + n = p + q$$

You can easily write context-free grammars for each of these languages.

(d)  $L = \{a, b\}^* - L_1$ , where  $L_1$  is the language  $\{b^m a^n b^m : m, n \geq 1\}$ . This one is interesting.  $L_1$  is not context free. But its complement  $L$  is. There are two ways to show this:

1. We could build a PDA. We can't build a PDA for  $L_1$ : if we count the first group of a's then we'll need to pop them to match against the second. But then what do we do for the third?  $L$  is easier though. We don't need to check that all the a groups are right. We simply have to find one that is wrong. We can do that with a nondeterministic pda  $P$ .  $P$  goes through the string making sure that the basic  $b(a^+b)^+$  structure is followed. Also, it nondeterministically chooses one group of a's to count. It then checks that the following group does not contain one more a. If any branch succeeds (i.e., it finds a mismatched pair of adjacent a groups) then  $P$  accepts.

2. We could use the same technique we used in (b) and (c) and decompose  $L$  into the union of two simpler languages. Just as we did in (b), we ask how a string could fail to be in  $L_1$  and thus be in  $L$ . The answer is that it could fail to have the correct  $b(a^+b)^+$  structure or it could have regions of a's that don't follow the rule that each region must contain one more a than did its predecessor. Thus  $L$  is the union of two languages:

- $L_2 = (a \cup b)^* - b(a^+b)^+$

- $L_3 = \{x b a^m b a^n b y \in \{a, b\}^* : m+1 \neq n\}$ .

It's easy to show that  $L_2$  is context free: Since  $b(a^+b)^+$  is regular its complement is regular and thus context free.  $L_3$  is also context free. You can build either a CFG or a PDA for it. It's very similar to the simpler language  $a^n b^m \ n \neq m$  that we used in (b) above. So  $L = L_2 \cup L_3$  is context free.

2. (a)  $L = \{a^{n^2} : n \geq 0\}$ . Suppose  $L = \{a^{n^2} : n \geq 0\}$  were context free. Then we could pump. Let  $n = M^2$ . So  $w$  is the string with  $M^{2^2}$ , or  $M^4$ , a's.) Clearly  $|w| \geq K$ , since  $M > K$ . So  $uvvxyyz$  must be in  $L$  (whatever  $v$  and  $y$

are). But it can't be. Why not? Given our  $w$ , the next element of  $L$  is the string with  $(M^2+1)^2$  a's. That's  $M^4 + 2M^2 + 1$  (expanding it out). But we know that  $|vxy| \leq M$ , so we can't pump in more than  $M$  a's when we pump only once. Thus the string we just created can't have more than  $M^4 + M$  a's. Clearly not enough.

(b)  $L = \{www : w \in \{a, b\}^*\}$ . The easiest way to do this is not to prove directly that  $L = \{www : w \in \{a, b\}^*\}$  is not context free. Instead, let's consider  $L1 = L \cap a^*ba^*ba^*b$ . If  $L$  is context free,  $L1$  must also be.  $L1 = \{a^nba^nba^n : n \geq 0\}$ . To show that  $L1$  is not context free, let's choose  $w = a^Mba^Mba^Mb$ . First we observe that neither  $v$  nor  $y$  can contain  $b$ , because if either did, then, when we pump, we'd have more than three b's, which is not allowed. So both must be in one of the three a regions. We consider the cases:

(1, 1) That group of a's will no longer match the other two, so the string is not in  $L1$ .

(2, 2) "

(3, 3) "

(1, 2) At least one of these two groups will have something pumped into it and will no longer match the one that is left out.

(2, 3) "

(1, 3) excluded since  $|vxy| \leq M$ , so  $vxy$  can't span the middle region of a's.

(c)  $L = \{w \in \{a, b, c\}^* : w \text{ has equal numbers of a's, b's, and c's}\}$ . Again, the easiest thing to do is first to intersect  $L = \{w \in \{a, b, c\}^* : w \text{ has equal numbers of a's, b's, and c's}\}$  with a regular language. This time we construct  $L1 = L \cap a^*b^*c^*$ .  $L1$  must be context free if  $L$  is. But  $L1 = a^n b^n c^n$ , which we've already proven is not context free. So  $L$  isn't either.

(d)  $L = \{a^n b^m a^n : n \geq m\}$ . We'll use the pumping lemma to show that  $L = \{a^n b^m a^n : n \geq m\}$  is not context free. Choose  $w = a^M b^M a^M$ . We know that neither  $v$  nor  $y$  can cross a and b regions, because if one of them did, then, when we pumped, we'd get a's and b's out of order. So we need only consider the cases where each is in one of the three regions of  $w$  (the first group of a's, the b's, and the second group of a's.)

(1, 1) The first group of a's will no longer match the second group.

(2, 2) If we pump in b's, then at some point there will be more b's than a's, and that's not allowed.

(3, 3) Analogous to (1, 1)

(1, 2) We must either (or both) pump a's into region 1, which means the two a regions won't match, or, if  $y$  is not empty, we'll pump in b's but then eventually there will be more b's than a's.

(2, 3) Analogous to (1, 2)

(1, 3) Ruled out since  $|vxy| \leq M$ , so  $vxy$  can't span the middle region of b's.

(e)  $L = \{a^n b^m c^n d^{(n+m)} : m, n \geq 0\}$ . We can show that  $L = \{a^n b^m c^n d^{(n+m)} : m, n \geq 0\}$  is not context free by pumping. We choose  $w = a^M b^M c^M d^{2M}$ . Clearly neither  $v$  nor  $y$  can cross regions and include more than one letter, since if that happened we'd get letters out of order when we pumped. So we only consider the cases where  $v$  and  $y$  fall within a single region. We'll consider four regions, corresponding to a, b, c, and d.

(1, 1) We'll change the number of a's and they won't match the c's any more.

(1, 2) If  $v$  is not empty, we'll change the a's and they won't match the c's. If  $y$  is nonempty, we'll change the number of b's and then we won't have the right number of d's any more.

(1, 3), (1, 4) are ruled out because  $|vxy| \leq M$ , so  $vxy$  can't span across any whole regions.

(2, 2) We'll change the number of b's but then we won't have the right number of d's.

(2, 3) If  $v$  is not empty, we'll change the b's without changing the d's. If  $y$  is not empty, we'll change the c's and they'll no longer match the a's.

(2, 4) is ruled out because  $|vxy| \leq M$ , so  $vxy$  can't span across any whole regions.

(3, 3) We'll change the number of c's and they won't match the a's.

(3, 4) If  $v$  is not empty, we'll change c's and they won't match a's. If  $y$  is not empty, we'll change d's without changing b's.

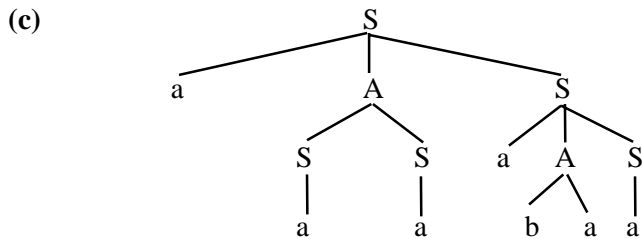
(4, 4) We'll change d's without changing a's or b's.

3. Let  $L = \{ a^n b^m c^p : n = m \text{ or } m = p \}$ .  $L$  is clearly context free. We can build a nondeterministic PDA  $M$  to accept it.  $M$  has two forks, one of which compares  $n$  to  $m$  and the other of which compares  $m$  to  $p$  (skipping over the  $a$ 's).  $L_1 = \{ a^n b^m c^p : n = m \text{ and } m = p \}$  is a subset of  $L$ . But  $L_1 = a^n b^n c^n$ , which we know is not context free.

4. (1) is fine. (2) is fine if we don't over interpret it. In particular, although both languages are defined in terms of the variable  $n$ , the scope of that variable is a single language. So within each individual language definition, the two occurrences of  $n$  are correctly interpreted to be occurrences of a single variable, and thus the values must be same both times. However, when we concatenate the two languages, we still have two separate language definitions with separate variables. So the two  $n$ 's are different. This is the key. It means that we can't assume that, given  $\{ a^n b^n \} \{ b^n a^n \}$ , we choose the same value of  $n$  for the two strings we choose. For example, we could get  $a^2 b^2 b^3 a^3$ , which is  $a^2 b^5 a^3$ , which is clearly not in  $\{ a^n b^{2n} a^n \}$ .

5. (a) (i) False, since all regular languages are also context free.  
(ii) True.  
(iii) False. For example  $a \in L$ , but is not long enough to contain pumpable substrings.  
(iv) False.  
(v) True, since the context-free languages are closed under reversal.

(b)  $S \Rightarrow aAS \Rightarrow aSSS \Rightarrow aaSS \Rightarrow aaaS \Rightarrow aaaaAS \Rightarrow aaaabaS \Rightarrow aaaabaa$ .



(d)  $M = (\{p, q\}, \{a, b\}, \{a, b\}, p, \{q\}, \Delta)$ , where  $\Delta =$   
 $\{((p, \epsilon, \epsilon), (q, S)), ((q, a, a), (q, \epsilon)), ((q, b, b), (q, \epsilon)), ((q, \epsilon, S), (q, aAS)), ((q, \epsilon, S), (q, a)),$   
 $((q, \epsilon, A), (q, SbA)), ((q, \epsilon, A), (q, SS)), ((q, \epsilon, A), (q, ba)) \}$

6. The easy way to show that  $L = \{ xx^R yy^R zz^R : x, y, z \in \{a, b\}^* \}$  is context free is to recall that we've already shown that  $\{ x x^R : x \in \{a, b\}^* \}$  is context free, and the context-free languages are closed under concatenation. But we can also do this directly by giving a grammar for  $L$ :

- $S \rightarrow AAA$
- $A \rightarrow aAa$
- $A \rightarrow bAb$
- $A \rightarrow \epsilon$

7. (a)  $L - R$  is context free.  $L - R = L \cap R'$  (the complement of  $R$ ).  $R'$  is regular (since the regular languages are closed under complement) and the intersection of a context-free language and a regular language is context-free, so  $L - R$  is context free.

(b)  $R - L$  need not be context free.  $R - L = R \cap L'$ . But  $L'$  may not be context free, since the context-free languages are not closed under complement. (The deterministic ones are, but  $L$  may not be deterministic.) If we let  $R = \Sigma^*$ , then  $R - L$  is exactly the complement of  $L$ .

8.  $M_1$ , which accepts  $L_1 = (\{1, 2\}, \{a, b\}, \{a\}, \Delta, 1, \{2\})$ ,  $\Delta =$   
 $((1, a, \epsilon), (1, a))$   
 $((1, b, a), (2, \epsilon))$   
 $((1, \epsilon, \epsilon), (2, \epsilon))$   
 $((2, b, a), (2, \epsilon))$

$((2, \varepsilon, a), (2, \varepsilon))$

$M_2$ , which accepts  $R_1 = (\{1, 2, 3, 4\}, \{a, b\}, \delta, 1, \{2\})$ ,  $\delta =$

$(1, a, 2)$

$(1, b, 3)$

$(2, a, 1)$

$(2, b, 4)$

$(3, a, 4)$

$(3, b, 1)$

$(4, a, 3)$

$(4, b, 2)$

$M_3$ , which accepts  $L_1 \cap R_1 = (\{(1, 1), (1, 2), (1, 3), (1, 4), (2, 1), (2, 2), (2, 3), (2, 4)\}, \{a, b\}, \{a\}, \Delta, (1,1), \{(2,2)\})$ ,  $\Delta =$

$((1, 1), a, \varepsilon), ((1, 2), a)$

$((2, 1), b, a), ((2, 3), \varepsilon)$

$((1, 1), \varepsilon, \varepsilon), ((2, 1), \varepsilon)$

$((1, 1), b, a), ((2, 3), \varepsilon)$

$((2, 2), b, a), ((2, 4), \varepsilon)$

$((1, 2), \varepsilon, \varepsilon), ((2, 2), \varepsilon)$

$((1, 2), a, \varepsilon), ((1, 1), a)$

$((2, 3), b, a), ((2, 1), \varepsilon)$

$((1, 3), \varepsilon, \varepsilon), ((2, 3), \varepsilon)$

$((1, 2), b, a), ((2, 4), \varepsilon)$

$((2, 4), b, a), ((2, 2), \varepsilon)$

$((1, 4), \varepsilon, \varepsilon), ((2, 4), \varepsilon)$

$((1, 3), a, \varepsilon), ((1, 4), a)$

$((2, 1), \varepsilon, a), ((2, 1), \varepsilon)$

$((1, 3), b, a), ((2, 1), \varepsilon)$

$((2, 2), \varepsilon, a), ((2, 2), \varepsilon)$

$((1, 4), a, \varepsilon), ((1, 3), a)$

$((2, 3), \varepsilon, a), ((2, 3), \varepsilon)$

$((1, 4), b, a), ((2, 2), \varepsilon)$

$((2, 4), \varepsilon, a), ((2, 4), \varepsilon)$

## CS 341 Homework 17 Turing Machines

1. Let  $M = (K, \Sigma, \delta, s, \{h\})$ , where

$$K = \{q_0, q_1, h\},$$

$$\Sigma = \{a, b, \square, \diamond\},$$

$$s = q_0,$$

and  $\delta$  is given by the following table,

$q$	$\sigma$	$\delta(q, \sigma)$
$q_0$	a	$(q_1, b)$
$q_0$	b	$(q_1, a)$
$q_0$	$\square$	$(h, \square)$
$q_0$	$\diamond$	$(q_0, \rightarrow)$
$q_1$	a	$(q_0, \rightarrow)$
$q_1$	b	$(q_0, \rightarrow)$
$q_1$	$\square$	$(q_0, \rightarrow)$
$q_1$	$\diamond$	$(q_1, \rightarrow)$

(a) Trace the computation of  $M$  starting from the configuration  $(q_0, \diamond \underline{a} b b b a)$ .

(b) Describe informally what  $M$  does when started in  $q_0$  on any square of a tape.

2. Repeat Problem 1 for the machine  $M = (K, \Sigma, \delta, s, \{h\})$ , where

$$K = \{q_0, q_1, q_2, h\},$$

$$\Sigma = \{a, b, \square, \diamond\},$$

$$s = q_0,$$

and  $\delta$  is given by the following table (the transitions on  $\diamond$  are  $\delta(q, \diamond) = (q, \diamond)$ , and are omitted).

$q$	$\sigma$	$\delta(q, \sigma)$
$q_0$	a	$(q_1, \leftarrow)$
$q_0$	b	$(q_0, \rightarrow)$
$q_0$	$\square$	$(q_0, \rightarrow)$
$q_1$	a	$(q_1, \leftarrow)$
$q_1$	b	$(q_2, \rightarrow)$
$q_1$	$\square$	$(q_1, \leftarrow)$
$q_2$	a	$(q_2, \rightarrow)$
$q_2$	b	$(q_2, \rightarrow)$
$q_2$	$\square$	$(h, \square)$

Start from the configuration  $(q_0, \diamond \underline{a} b b \square b b \square \square \square \underline{a} b a)$ .

3. Let  $M$  be the Turing machine  $M = (K, \Sigma, \delta, s, \{h\})$ , where

$$K = \{q_0, q_1, q_2, h\},$$

$$\Sigma = \{a, \square, \diamond\},$$

$$s = q_0,$$

and  $\delta$  is given by the following table.

Let  $n \geq 0$ . Describe carefully what  $M$  does when started in the configuration  $(q_0, \diamond \square a^n \underline{a})$ .



q	$\sigma$	$\delta(q, \sigma)$
q <sub>0</sub>	a	(q <sub>1</sub> , ←)
q <sub>0</sub>	□	(q <sub>0</sub> , □)
q <sub>0</sub>	◇	(q <sub>0</sub> , →)
q <sub>1</sub>	a	(q <sub>2</sub> , □)
q <sub>1</sub>	□	(h, □)
q <sub>1</sub>	◇	(q <sub>1</sub> , →)
q <sub>2</sub>	a	(q <sub>2</sub> , a)
q <sub>2</sub>	□	(q <sub>0</sub> , ←)
q <sub>2</sub>	◇	(q <sub>2</sub> , →)

4. Design and write out in full a Turing machine that scans to the right until it finds two consecutive a's and then halts. The alphabet of the Turing machine should be {a, b, □, ◇}.

5. Give a Turing machine (in our abbreviated notation) that takes as input a string  $w \in \{a, b\}^*$  and squeezes out the a's. Assume that the input configuration is  $(s, \diamond \square w)$  and the output configuration is  $(h, \diamond \square w')$ , where  $w' = w$  with all the a's removed.

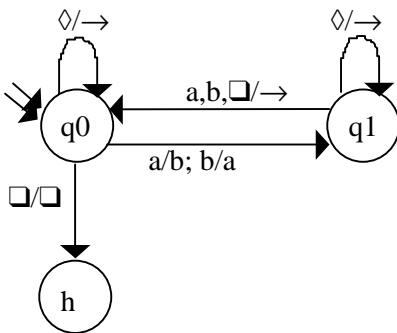
6. Give a Turing machine (in our abbreviated notation) that shifts its input two characters to the right.

Input:  $\square w \square$   
Output:  $\square \square w \square$

7. (L & P 5.7.2) Show that if a language is recursively enumerable, then there is a Turing machine that enumerates it without ever repeating an element of the language.

### Solutions

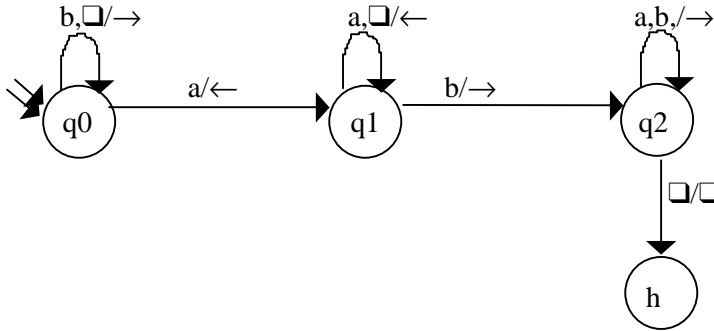
1. (a)



q<sub>0</sub>, ◇aabbba  
q<sub>1</sub>, ◇babbba  
q<sub>0</sub>, ◇babbba  
q<sub>1</sub>, ◇bbbbbba  
q<sub>0</sub>, ◇bbbbbbba  
q<sub>1</sub>, ◇bbabbba  
q<sub>0</sub>, ◇bbabba  
q<sub>1</sub>, ◇bbaaba  
q<sub>0</sub>, ◇bbaaba  
q<sub>1</sub>, ◇bbaaaa  
q<sub>0</sub>, ◇bbaaaaa  
q<sub>1</sub>, ◇bbaaaab  
q<sub>0</sub>, ◇bbaaab□  
h, ◇bbaaab□

(b) Converts all a's to b's, and vice versa, starting with the current symbol and moving right.

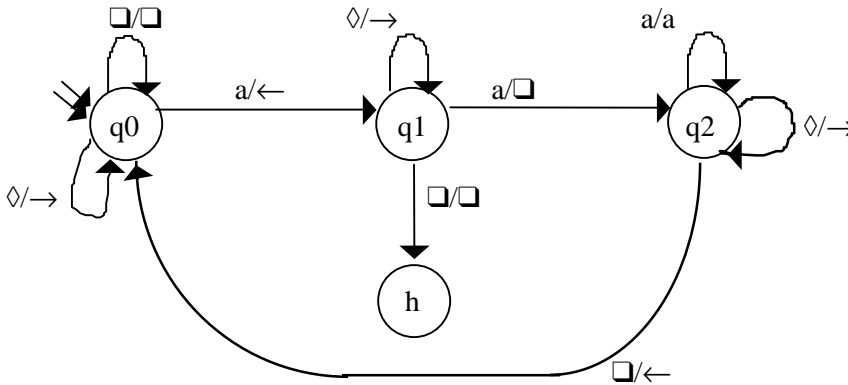
2. (a)



$q_0, \diamond \underline{a} \underline{b} \square \underline{b} \square \square \square \underline{a} \underline{b} \underline{a}$   
 $q_0, \diamond \underline{a} \underline{b} \square \underline{b} \square \square \square \underline{a} \underline{b} \underline{a}$   
 $q_0, \diamond \underline{a} \underline{b} \square \underline{b} \square \square \square \underline{a} \underline{b} \underline{a} \dots$   
 $q_1, \diamond \underline{a} \underline{b} \square \underline{b} \square \square \square \underline{a} \underline{b} \underline{a}$   
 $q_1, \diamond \underline{a} \underline{b} \square \underline{b} \square \square \square \underline{a} \underline{b} \underline{a} \dots$   
 $q_2, \diamond \underline{a} \underline{b} \square \underline{b} \square \square \square \underline{a} \underline{b} \underline{a}$   
 $h, \diamond \underline{a} \underline{b} \square \underline{b} \square \square \square \underline{a} \underline{b} \underline{a}$

(b) M goes right until it finds an a, then left until it finds a b, then right until it finds a blank.

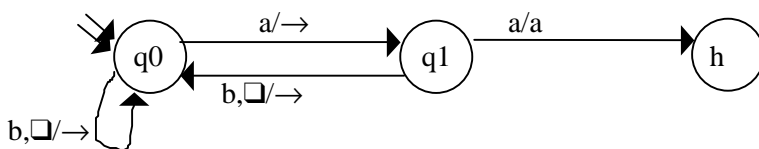
3.



$q_0, \diamond \underline{a} \underline{a} \underline{a} \underline{a} \underline{a}$   
 $q_1, \diamond \underline{a} \underline{a} \underline{a} \underline{a} \underline{a}$   
 $q_2, \diamond \underline{a} \underline{a} \underline{a} \underline{a} \underline{a}$   
 $q_0, \diamond \underline{a} \underline{a} \underline{a} \underline{a} \underline{a}$   
 $q_1, \diamond \underline{a} \underline{a} \underline{a} \underline{a} \underline{a}$   
 $q_2, \diamond \underline{a} \underline{a} \underline{a} \underline{a} \underline{a}$   
 $q_0, \diamond \underline{a} \underline{a} \underline{a} \underline{a} \underline{a}$   
 $q_1, \diamond \underline{a} \underline{a} \underline{a} \underline{a} \underline{a}$   
 $h, \diamond \underline{a} \underline{a} \underline{a} \underline{a} \underline{a}$

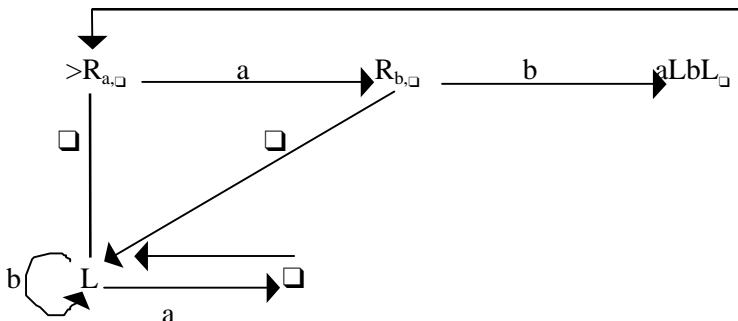
M changes every other a, moving left from the start, to a blank. If n is odd, it loops. If n is even, it halts.

4.

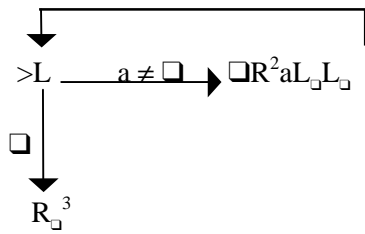


$M = (K, \Sigma, \delta, s, \{h\})$ , where  
 $K = \{q_0, q_1, h\}$ ,  
 $\Sigma = \{a, b, \square, \diamond\}$ ,  
 $s = q_0$

5. The idea here is that first we'll push all b's to the left, thus squeezing all the a's to the right. Then we'll just replace the a's with blanks. In more detail: scan the string from left to right. Every time we find an a, if there are any b's to the right of it we need to shift them left. So scan right, skipping as many a's as there are. When we find a b, swap it with the last a. That squeezes one a further to the right. Go back to the left end and repeat. Eventually all the a's will come after all the b's. At that point, when we look for a b following an a, all we'll find is a blank. At that point, we just clean up by rewriting all the a's as blanks.



6. The idea is to start with the rightmost character of  $w$ , rewrite it as a blank, then move two squares to the right and plunk that character back down. Then scan left for the next leftmost character, do the same thing, and so forth.



7. Suppose that  $M$  is the Turing machine that enumerates  $L$ . Then construct  $M^*$  to enumerate  $L$  with no repetitions:  $M^*$  will begin by simulating  $M$ . But whenever  $M$  outputs a string,  $M^*$  will first check to see if the string has been output before (see below). If it has, it will just continue looking for strings. If not, it will output it, and it will also write the string, with # in front of it, at the right end of the tape. To check whether a string has been output before,  $M^*$  just scans its tape checking for the string it is about to output.

## CS 341 Homework 18

### Computing with Turing Machines

1. Present Turing machines that decide the following languages over  $\{a, b\}$ :

- (a)  $\emptyset$
- (b)  $\{\epsilon\}$
- (c)  $\{a\}$
- (d)  $\{a\}^*$

2. Consider the simple (regular, in fact) language  $L = \{w \in \{a,b\}^* : |w| \text{ is even}\}$

- (a) Give a Turing machine that decides  $L$ .
- (b) Give a Turing machine that semidecides  $L$ .

3. Give a Turing machine (in our abbreviated notation) that accepts  $L = \{a^n b^m a^n : m > n\}$

4. Give a Turing machine (in our abbreviated notation) that accepts  $L = \{ww : w \in \{a, b\}^*\}$

5. Give a Turing machine (in our abbreviated notation) that computes the following function from strings in  $\{a, b\}^*$  to strings in  $\{a, b\}^* : f(w) = ww^R$ .

6. Give a Turing machine that computes the function  $f: \{a,b,c\}^* \rightarrow \mathbb{N}$  (the integers), where  $f(w)$  = the number of a's (in unary) in  $w$ .

7. Let  $w$  and  $x$  be any two positive integers encoded in unary. Show a Turing machine  $M$  that computes

$$f(w, x) = w + x.$$

Represent the input to  $M$  as

$$\diamond \square^w ; x \square$$

8. Two's complement form provides a way to represent both positive and negative binary integers. Suppose that the number of bits allocated to each number is  $k$  (generally the word size). Then each positive integer is represented simply as its binary encoding, with leading zeros. Each negative integer  $n$  is represented as the result of subtracting  $|n|$  from  $2^k$ , where  $k$  is the number of bits to be used in the representation. Given a fixed  $k$ , it is possible to represent any integer  $n$  if  $-2^{k-1} \leq n \leq 2^{k-1} - 1$ . The high order digit of each number indicates its sign: it is zero for positive integers and 1 for negative integers.

Examples, for  $k = 4$ :

$$0 = 0000, 1 = 0001, 2 = 0010, 3 = 0011, 4 = 0100, 5 = 0101, 6 = 0110, 7 = 0111$$

$$-1 = 1111, -2 = 1110, -3 = 1101, -4 = 1100, -5 = 1011, -6 = 1010, -7 = 1001, -8 = 1000$$

Since Turing machines don't have fixed length words, we'd like to be able to represent any integer. We will represent positive integers with a single leading 0. We will represent each negative integer  $n$  as the result of subtracting  $n$  from  $2^{i+1}$ , where  $i$  is the smallest value such that  $2^i \geq |n|$ . For example,  $-65$  will be represented as  $1111111$ , since  $2^7$  (128)  $\geq 65$ , so we subtract 65 (01000001 in binary) from  $2^8$  (in binary, 100000000). We need the extra digit (i.e., we subtract from  $2^{i+1}$  rather than from  $2^i$ ) because, in order for a positive number to be interpreted as positive, it must have a leading 0, thus consuming an extra digit.

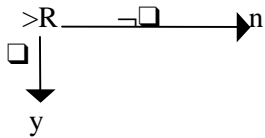
Let  $w$  be any integer encoded in two's complement form. Show a Turing machine that computes  $f(w) = -w$ .

**Solutions**

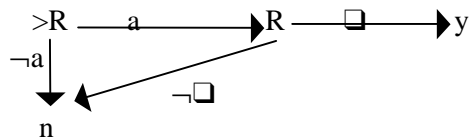
1 (a) We should reject everything, since no strings are in the language.

> n

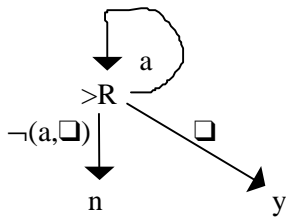
(b) Other than the left boundary symbol, the tape should be blank:  $\diamond \square \square \square$



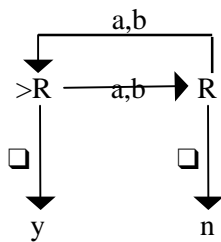
(c) Just the single string a:  $\diamond \square a \square$



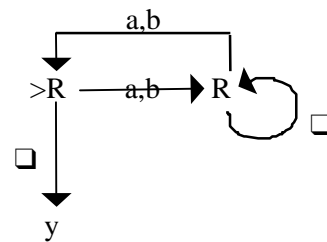
(d) Any number of a's:



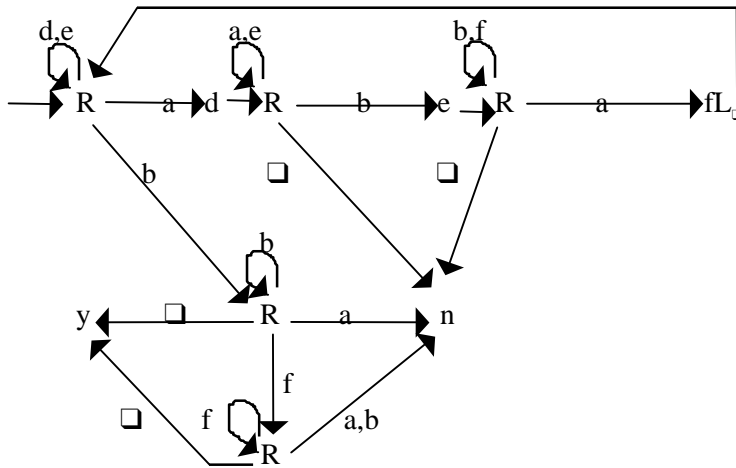
2. (a)



(b)



3. The idea is to make a sequence of passes over the input. On each pass, we mark off (with d, e, and f) a matching a, b, and a. This corresponds to the top row of the machine shown here. When there are no matching groups left, then we accept if there is nothing left or if there are b's in the middle. If there is anything else, we reject. It turns out that a great deal of this machine is essentially error checking. We get to the R on the second row as soon as we find the first "extra" b. We can loop in it as long as we find b's. If we find a's we reject. If we find a blank, then the string had just b's, which is okay, so we accept. Once we find an f, we have to go to the separate state R on the third row to skip over the f's and make sure we get to the final blank without either any more b's or any more a's.

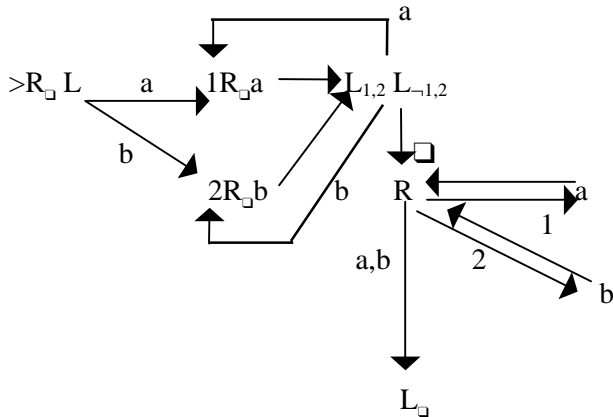


4. The hard part here is that we don't know where the middle of the string is. So we don't know where the boundary between the first occurrence of w ends and the second begins. We can break this problem into three subroutines, which will be executed in order:

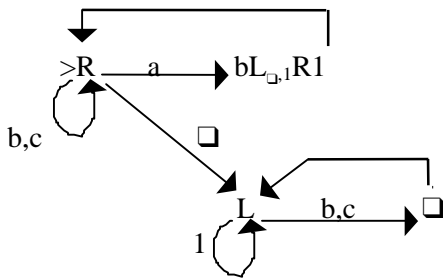
- (1) Find the middle and mark it. If there's a lone character in the middle (i.e., the length of the input string isn't even), then reject immediately.
- (2) Bounce back and forth between the beginning of the first w and the beginning of the second, marking off characters if they match and rejecting if they don't.
- (3) If we get to the end of the w's and everything has matched, accept.

Let's say a little more about step (1). We need to put a marker in the middle of the string. The easiest thing to do is to make a double marker. We'll use ##. That way, we can start at both ends (bouncing back and forth), moving a marker one character toward the middle at each step. For example, if we start with the tape  $\diamond \square aabbaabb \square \square \square$ , after one mark off step we'll have  $\diamond \square a \# abbaab \# b \square \square \square$ , then  $\diamond \square aa \# bbaa \# bb \square \square \square$ , and finally  $\diamond \square aabb \# \# aabb \square \square \square$ . So first we shift the whole input string two squares to the right on the tape to make room for the two markers. Then we bounce back and forth, moving the two markers toward the center. If they meet, we've got an even length string and we can continue. If they don't, we reject right away.

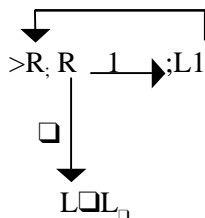
5. The idea is to work from the middle. We'll scan right to the rightmost character of  $w$  (which we find by scanning for the first blank, then backing up (left) one square. We'll rewrite it so we know not to deal with it again (a's will become 1's; b's will become 2's.) Then we move right and copy it. Now if we scan back left past any 1's or 2's, we'll find the next rightmost character of  $w$ . We rewrite it to a 1 or a 2, then scan right to a blank and copy it. We keep this up until, when we scan back to the left, past any 1's or 2's, we hit a blank. That means we've copied everything. Now we scan to the right, replacing 1's by a's and 2's by b's. Finally, we scan back to the left to position the read head to the left of  $w$ .



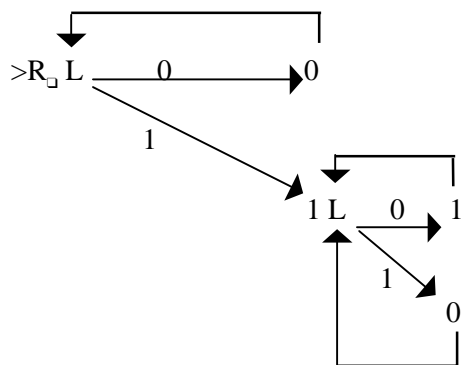
6. The idea here is that we need to write a 1 for every a and we can throw away b's and c's. We want the 1's to end up at the left end of the string, so then all we have to do to clean up at the end is erase all the b's and c's to the right of the area with the 1's. So, we'll start at the left edge of the string. We'll skip over b's and c's until we get to an a. At that point, rewrite it as b so we don't count it again. Then (remembering it in the state) scan left until we get to the blank (if this is the first 1) or we get to a 1. In either case, move one square to the right and write a 1. We are thus overwriting a b or a c, but we don't care. We're going to throw them away anyway. Now start again scanning to the right looking for an a. At some point, we'll come to the end of string blank instead. At that point, just travel leftward, rewriting all the b's and c's to blank, then cross the 1's and land on the blank at the left of the string.



7. All we have to do is to concatenate the two strings. So shift the second one left one square, covering up the semicolon.



8. Do a couple of examples of the conversion to see what's going on. What you'll observe is that we want to scan from the right. Initially, we may see some zeros, and those will stay as zeros. If we ever see a 1, then we rewrite the first one as a 1. After that, we're dealing with borrowing, so we swap all digits: every zero becomes a one and every one becomes a zero, until we hit the blank at the end of the string and halt.





## CS 341 Homework 19

### Turing Machine Extensions

1. Consider the language  $L = \{ww^R\}$ .

(a) Describe a one tape Turing machine to accept  $L$ .

(b) Describe a two tape Turing machine to accept  $L$ .

(c) How much more efficient is the two tape machine?

2. Give (in abbreviated notation) a nondeterministic Turing machine that accepts the language

$$L = \{ww^Ruu^R : w, u \in \{a, b\}^*\}$$

### Solutions

(1) (a) The one tape machine needs to bounce back and forth between the beginning of the input string and the end, marking off matching symbols.

(b) The two tape machine works as follows: If the input is  $\epsilon$ , accept. If not, copy the input to the second tape and record in the state that you have processed an even number of characters so far. Now, start the first tape at the left end and the second tape at the right end. Check that the symbols on the two tapes are the same. If not, reject. If so, move the first tape head to the right and the second tape head to the left. Also record that you have processed an odd number and continue, each time using the state to keep track of whether you've seen an even or odd number of characters so far. When you reach the end of the input tape, accept if you've seen an even number of characters. Reject if you've seen an odd number. (The even/odd counter is necessary to make sure that you reject strings such as  $aba$ .)

(c) The one tape machine takes time proportional to the square of the length of the input, since for an input of length  $n$  it will make  $n$  passes over the input, each of which takes on average  $n/2$  steps. The two tape machine takes time that's linear in  $n$ . It takes  $n$  steps to copy, then another  $n$  steps to compare.

2. The idea is just to use nondeterminism to guess the location of the boundary between the  $w$  and  $u$  regions. Each path will choose a spot, shift the  $u$  region to the right, and insert a boundary marker  $\#$ . Once this is done, the machine simply checks each region for  $ww^R$ . If we get a string in  $L$ , one of the guessed paths will work.

## CS 341 Homework 20 Unrestricted Grammars

1. Find grammars that generate the following languages:

(a)  $L = \{ww : w \in \{a, b\}^*\}$

(b)  $L = \{a^{2^n} : n \geq 0\}$

(c)  $L = \{a^n b^{2n} c^{3n} : n \geq 1\}$

(d)  $L = \{w^R : w \text{ is the social security number of a living American citizen}\}$

(e)  $L = \{wc^m d^n : w \in \{a, b\}^* \text{ and } m = \text{the number of a's in } w \text{ and } n \text{ equals the number of b's in } w\}$

2. Find a grammar that computes the function  $f(w) = ww$ , where  $w \in \{a, b\}^*$ .

### Solutions

1. (a)  $L = \{ww : w \in \{a, b\}^*\}$

There isn't any way to generate the two  $w$ 's in the correct order. Suppose we try. Then we could get  $aSa$ . Suppose we want  $b$  next. Then we need  $Sa$  to become  $bSab$ , since the new  $b$  has to come after the  $a$  that's already there. That could work. Now we have  $abSab$ . Let's say we want  $a$  next. Now  $Sab$  has to become  $aSaba$ . The problem is that, as the length of the string grows, so does the number of rules we'll need to cope with all the patterns we could have to replace. In a finite number of rules, we can't deal with replacing  $S$  (which we need to do to get the next character in the first occurrence of  $w$ ), and adding a new character that is arbitrarily far away from  $S$ .

The other approach we could try would be to have a rule  $S \rightarrow WW$ , and then let  $W$  generate a string of  $a$ 's and  $b$ 's. But this won't work, since we have no way to control the expansion of the two  $W$ 's so that they produce the same thing.

So what we need to do is to generate  $ww^R$  and then, carefully, reverse the order of the characters in  $w^R$ . What we'll do is to start by erecting a wall ( $\#$ ) at the right end of the string. Then we'll generate  $ww^R$ . Then, in a second phase, we'll take the characters in the second  $w$  and, one at a time, starting with the leftmost, move it right and then move it past the wall. At each step, we move each character up to the wall and then just over it, but we don't reverse characters once they get over the wall. The first part of the grammar, which will generate  $wTw^R$ , looks like this:

$S \rightarrow S_1 \#$       This inserts the wall at the right.

$S_1 \rightarrow aS_1a$

$S_1 \rightarrow bS_1b$

$S_1 \rightarrow T$        $T$  will mark the left edge of the portion that needs to be reversed.

At this point, we can generate strings such as  $abbbTbbba\#$ . What we need to do now is to reverse the string of  $a$ 's and  $b$ 's that is between  $T$  and  $\#$ . To do that, we let  $T$  spin off a marker  $Q$ , which we can pass rightward through the string. As it moves to the right, it will take the first  $a$  or  $b$  it finds with it. It does this by swapping the character it is carrying (the one just to the right of it) with the next one to the right. It also moves itself one square to the right. The four rules marked with  $*$  accomplish this. When  $Q$ 's character gets to the  $\#$  (the rules marked  $**$ ), the  $a$  or  $b$  will swap places with the  $\#$  (thus hopping the fence) and the  $Q$  will go away. We can keep doing this until all the  $a$ 's and  $b$ 's are behind the fence and in the right order. Then the final  $T\#$  will drop out. Here are the rules for this phase:

$T \rightarrow TQ$   
 $Qaa \rightarrow aQa$      \*  
 $Qab \rightarrow bQa$      \*  
 $Qbb \rightarrow bQb$      \*  
 $Qba \rightarrow aQb$      \*  
 $Qa\# \rightarrow \#a$      \*\*  
 $Qb\# \rightarrow \#b$      \*\*  
 $T\# \rightarrow \epsilon$

So with R as given above, the grammar  $G = (\{S, S_1, \#, T, Q, a, b\}, \{a, b\}, R, S)$

**(b)**  $L = \{ a^{2^n} : n \geq 0 \}$

The idea here is first to generate the first string, which is just a. Then think about the next one. You can derive it by taking the previous one, and, for every a, write two a's. So we get aa. Now to get the third one, we do the same thing. Each of the two a's becomes two and we have four, and so forth. So we need a rule to get us started and to indicate the possibility of duplication. Then we need rules to actually do the duplication. To make duplication happen, we need a symbol that gets generated by S indicating the option to repeat. We'll use P. Since duplication can happen an arbitrary number of times, we need P to spin off as many individual duplication commands as we want. We'll use R for that. The one other thing we need is to make sure, if we start a duplication step, that we finish it. In other words, suppose we currently have aaaa. If we start duplicating the a's, we must duplicate all of them. Otherwise, we might end up with, for example, seven a's. So we'll introduce a left edge marker, #. Once we fire up a duplication (by creating an R), we'll only stop (i.e., get rid of R) when R has made it all the way to the other end of the string (namely the left end since it starts at the right). So we get the following rules:

$S \rightarrow \#aP$	P lets us start up duplication processes as often as we like.
$P \rightarrow \epsilon$	When we've done as many as we want, we get rid of P.
$P \rightarrow RP$	R will actually do a duplication by moving leftward, duplicating every a it sees.
$aR \rightarrow Raa$	Actually duplicates one a, and moves R one square to the left so it moves on to the next a
$\#R \rightarrow \#$	Get rid of R once it's made it all the way to the left
$\# \rightarrow \epsilon$	Get of # at the end

So with R as given above, the grammar  $G = (\{S, P, R, \#, a, b\}, \{a, b\}, R, S)$

**(c)**  $L = \{ a^n b^{2n} c^{3n} : n \geq 1 \}$

This one is very similar to  $a^n b^n c^n$ . The only difference is that we will churn out b's in pairs and c's in triples each time we expand S. So we get:

$S \rightarrow aBSccc$   
 $S \rightarrow aBccc$   
 $Ba \rightarrow aB$   
 $Bc \rightarrow bbc$   
 $Bb \rightarrow bbb$

So with R as given above, the grammar  $G = (\{S, B, a, b, c\}, \{a, b, c\}, R, S)$

**(d)**  $L = \{ w^R : w \text{ is the social security number of a living American citizen} \}$

This one is regular. There is a finite number of such social security numbers. So we need one rule for each number. Each rule is of the form  $S \rightarrow \langle \text{valid number} \rangle$ . So with that collection of rules as R, the grammar  $G = (\{S, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, R, S)$

**(e)**  $L = \{ wc^m d^n : w \in \{a, b\}^* \text{ and } m = \text{the number of a's in } w \text{ and } n \text{ equals the number of b's in } w \}$

The idea here is to generate a c every time we generate an a and to generate a d every time we generate a b. We'll do this by generating the nonterminals C and D, which we will use to generate c's and d's once everything is in the right place. Once we've finished generating all the a's and b's we want, the next thing we need to do is to get

all the D's to the far right of the string, all the C's next, and then have the a's and b's left alone at the left. We guarantee that everything must line up that way by making sure that C can't become c and D can't become d unless things are right. To do this, we require that D can only become d if it's all the way to the right (i.e., it's followed by #) or it's got a d to its right. Similarly with C. We can do this with the following rules;

- $S \rightarrow S_1\#$
- $S_1 \rightarrow aS_1C$
- $S_1 \rightarrow bS_1D$
- $S_1 \rightarrow \epsilon$
- $DC \rightarrow CD$
- $D\# \rightarrow d$
- $Dd \rightarrow dd$
- $C\# \rightarrow c$
- $Cd \rightarrow cd$
- $Cc \rightarrow cc$
- $\# \rightarrow \epsilon$

So with R as given above, the grammar  $G = (\{S, S_1, C, D, \#, a, b, c, d\}, \{a, b, c, d\}, R, S)$

2. We need to find a grammar that computes the function  $f(w) = ww$ . So we'll get inputs such as SabaS. Think of the grammar we'll build as a procedure, which will work as described below. At any given time, the string that has just been derived will be composed of the following regions:

<the part of w that has already been inserted copied>	S	<the part of w that has not yet been copied, which may have within it a character (preceded by #) that is currently being copied by being moved through the region>	T (inserted when the first character moves into the copy region)	<the part of the second w that has been copied so far, which may have within it a character (preceded by %) that is currently being moved through the region>	W (also when T is)
---	---	---	--	---	--------------------

Most of the rules come in pairs, one dealing with an a, the other with b.

- $SS \rightarrow \epsilon$  Handles the empty string.
- $Sa \rightarrow aS\#a$  Move S past the first a to indicate that it has already been copied. Then start copying it by introducing a new a, preceded by the special marker #, which we'll use to push the new a to the right end of the string.
- $Sb \rightarrow bS\#b$  Same for copying b.
- $\#aa \rightarrow a\#a$  Move the a we're copying past the next character if it's an a.
- $\#ab \rightarrow b\#a$  Move the a we're copying past the next character if it's a b.
- $\#ba \rightarrow a\#b$  Same two rules for pushing b.
- $\#bb \rightarrow b\#b$  "
- $\#aS \rightarrow \#aTW$  We've gotten to the end of w. This is the first character to be copied, so the initial S is at the end of w. We need to create a boundary between w and the copied w. T will be that boundary. We also need to create a boundary for the end of the copied w. W will be that boundary. T and W are adjacent at this point because we haven't copied any characters into the copy region yet.
- $\#bS \rightarrow \#aTW$  Same if we get to the end of w pushing b.
- $\#aT \rightarrow T\%a$  Jump the a we're copying into the copy region (i.e., to the right of T). Get rid of #, since we're done with it. Introduce %, which we'll use to push the copied a through the copy region.
- $\#bT \rightarrow T\%b$  Same if we're pushing b.

$\%aa \rightarrow a\%a$  Push a to the right through the copied region in exactly the same way we pushed it through w, except we're using % rather than # as the pusher. This rule pushes a past a.  
 $\%ab \rightarrow b\%a$  Pushes a past b.  
 $\%ba \rightarrow a\%b$  Same two rules for pushing b.  
 $\%bb \rightarrow b\%b$  "  
 $\%aW \rightarrow aW$  We've pushed an a all the way to the right boundary, so get rid of %, the pusher.  
 $\%bW \rightarrow bW$  Same for a pushed b.  
 $ST \rightarrow \epsilon$  All the characters from w have been copied, so they're all to the left of S, which causes S to be adjacent to the middle marker T. We can now get rid of our special walls. Here we get rid of S and T.  
 $W \rightarrow \epsilon$  Get rid of W. Note that if we do this before we should, there's no way to get rid of %, so any derivation path that does this will fail to produce a string in  $\{a, b\}^*$ .

So with R as given above, the grammar  $G = (\{S, T, W, \#, \%, a, b\}, \{a, b\}, R, S)$

## CS 341 Homework 21

### Undecidability

1. Which of the following problems about Turing machines are solvable, and which are undecidable? Explain your answers carefully.

- (a) To determine, given a Turing machine  $M$ , a state  $q$ , and a string  $w$ , whether  $M$  ever reaches state  $q$  when started with input  $w$  from its initial state.
- (b) To determine, given a Turing machine  $M$  and a string  $w$ , whether  $M$  ever moves its head to the left when started with input  $w$ .
- (c) To determine, given two Turing machines, whether one semidecides the complement of the language semidecided by the other.
- (d) To determine, given a Turing machine  $M$ , whether the language semidecided by  $M$  is finite.

2. Show that it is decidable, given a pushdown automaton  $M$  with one state, whether  $L(M) = \Sigma^*$ . (Hint: Show that such an automaton accepts all strings if and only if it accepts all strings of length one.)

3. Which of the following problems about context-free grammars are solvable, and which are undecidable? Explain your answers carefully.

- (a) To determine, given a context-free grammar  $G$ , is  $\epsilon \in L(G)$ ?
- (b) To determine, given a context-free grammar  $G$ , is  $\{\epsilon\} = L(G)$ ?
- (c) To determine, given two context-free grammars  $G_1$  and  $G_2$ , is  $L(G_1) \subseteq L(G_2)$ ?

4. The nonrecursive languages  $L$  that we have discussed in class all have the property that either  $L$  or the complement of  $L$  is recursively enumerable.

- (a) Show by a counting argument that there is a language  $L$  such that neither  $L$  nor its complement is recursively enumerable.
- (b) Give an example of such a language.

### Solutions

1. (a) To determine, given a Turing machine  $M$ , a state  $q$ , and a string  $w$ , whether  $M$  ever reaches state  $q$  when started with input  $w$  from its initial state. This is not solvable. We can reduce  $H$  to it. Essentially, if we can tell whether a machine  $M$  ever reaches some state  $q$ , then let  $q$  be  $M$ 's halt state (and we can massage  $M$  so it has only one halt state). If it ever gets to  $q$ , it must have halted. More formally:

$$L_1 = H = \{s = "M" "w" : M \text{ halts on input string } w\}$$

$$\Downarrow \quad \tau$$

$$(?M_2) \quad L_2 = \{s : "M" "w" "q" : M \text{ reaches state } q \text{ when started with input } w \text{ from its initial state}\}$$

Let  $\tau'$  create, from  $M$  the machine  $M^*$  as follows. Initially  $M^*$  equals  $M$ . Next, a new halting state  $H$  is created in  $M^*$ . Then, from each state that was a halting state in  $M$ , we create transitions in  $M^*$  such that for all possible values of the current tape square,  $M^*$  goes to  $H$ . We create no other transitions to  $H$ . Notice that  $M^*$  will end up in  $H$  in precisely the same situations in which  $M$  halts.

$$\text{Now let } \tau("M" "w") = \tau'("M") "w" "H"$$

So, if  $M_2$  exists, then  $M_1$  exists. It invokes  $\tau'$  to create  $M^*$ . Then it passes " $M^*$ ", " $w$ ", and " $H$ " to  $M_2$  and returns whatever  $M_2$  returns. But  $M_1$  doesn't exist. So neither does  $M_2$ .

(b) To determine, given a Turing machine  $M$  and a string  $w$ , whether  $M$  ever moves its head to the left when started with input  $w$ . This one is solvable. We will assume that  $M$  is deterministic. We can build the deciding machine  $D$  as follows.  $D$  starts by simulating the operation of  $M$  on  $w$ .  $D$  keeps track on another tape of each configuration of  $M$  that it has seen so far. Eventually, one of the following things must happen:

1.  $M$  moves its head to the left. In this case, we say yes.
2.  $M$  is stuck on some square  $s$  of the tape. In other words, it is in some state  $p$  looking at some square  $s$  on the tape and it has been in this configuration before. If this happens and  $M$  didn't go left yet, then  $M$  simply hasn't moved off of  $s$ . And it won't from now on, since it's just going to do the same thing at this point as it did the last time it was in this configuration. So we say no.
3.  $M$  moves off the right hand edge of the input  $w$ . So it is in some state  $p$  looking at a blank. Within  $k$  steps (if  $k$  is the number of states in  $M$ ),  $M$  must repeat some state  $p$ . If it does this without moving left, then again we know that it never will. In other words, if the last time it was in the configuration in which it was in state  $p$ , looking at a blank, there was nothing to the right except blanks, and it can't move left, and it is again in that same situation, it will do exactly the same thing again. So we say no.

(c) To determine, given two Turing machines, whether one semidecides the complement of the language semidecided by the other. This one is not solvable. We can reduce to it the problem, "Given a Turing machine  $M$ , is there any string at all on which  $M$  halts?" (Which is equivalent to "Is  $L(M) = \emptyset$ ?") In the book we show that this problem is not solvable. What we'll do is to build a machine  $M^*$  that semidecides the language  $\Sigma^*$ , which is the complement of the language  $\emptyset$ . If we could build a machine to tell, given two Turing machines, whether one semidecides the complement of the language semidecided by the other, then to find out whether any given machine  $M$  accepts anything, we'd pass  $M$  and our constructed  $M^*$  to this new machine. If it says yes, then  $M$  accepts  $\emptyset$ . If it says no, then  $M$  must accept something. Formally:

$$L_1 = \{s = "M" \mid M \text{ halts on some string } w\}$$

$$\Downarrow \quad \tau$$

$$(?M_2) \quad L_2 = \{s = "M_1" "M_2" : M_1 \text{ decides the complement of the language semidecided by } M_2\}$$

$M$  accepts strings over some input alphabet  $\Sigma$ . Let  $\tau'$  construct a machine  $M^*$  that semidecides the language  $\Sigma^*$ . Then  $\tau("M") = "M" " \tau'(M)"$ .

So, if  $M_2$  exists, then  $M_1$  exists. It invokes  $\tau'$  to create  $M^*$ . Then it passes " $M$ " and " $M^*$ " to  $M_2$  and returns the opposite of whatever  $M_2$  returns (since  $M_2$  says yes if  $L(M) = \emptyset$  and  $M_1$  wants to say yes if  $L(M) \neq \emptyset$ ). But  $M_1$  doesn't exist. So neither does  $M_2$ .

(d) To determine, given a Turing machine  $M$ , whether the language semidecided by  $M$  is finite. This one isn't solvable. We can reduce to it the problem, "Given a Turing machine  $M$ , does  $M$  halt on  $\epsilon$ ?" We'll construct, from  $M$ , a new machine  $M^*$ , which erases its input tape and then simulates  $M$ .  $M^*$  halts on all inputs iff  $M$  halts on  $\epsilon$ . If  $M$  doesn't halt on  $\epsilon$ , then  $M^*$  halts on no inputs. So there are two situations:  $M^*$  halts on all inputs (i.e.,  $L(M^*)$  is infinite) or  $M^*$  halts on no inputs (i.e.,  $L(M^*)$  is finite). So, if we could build a Turing machine  $M_2$  to decide whether  $L(M^*)$  is finite or infinite, we could build a machine  $M_1$  to decide whether  $M$  halts on  $\epsilon$ . Formally:

$$L_1 = \{s = "M" \mid M \text{ halts on } \epsilon\}$$

$$\Downarrow \quad \tau$$

$$(?M_2) \quad L_2 = \{s = "M" \mid \text{is finite}\}$$

Let  $\tau$  construct the machine  $M^*$  from "M" as described above.

So, if  $M_2$  exists, then  $M_1$  exists. It invokes  $\tau$  to create  $M^*$  which accepts a finite language precisely if  $M$  accepts  $\epsilon$ . But  $M_1$  doesn't exist. So neither does  $M_2$ .

**2.**  $M$  only has one state  $S$ . If  $S$  is not a final state, then  $L(M) = \emptyset$ , which is clearly not equal to  $\Sigma^*$ , so we say no. Now suppose that  $S$  is a final state. Then  $M$  accepts  $\epsilon$ . Does it also accept anything else? To accept any single character  $c$  in  $\Sigma$ , there must be a transition  $((S, c, \epsilon), (S, \epsilon))$ . In other words, we must be able to end up in  $S$  with an empty stack if, looking at an empty stack, we see  $c$ . If there is not such a transition for every element  $c$  of  $\Sigma$ , then we say no, since we clearly cannot get even all the one character strings in  $\Sigma^*$ . Now, suppose that all those required transitions do exist. Then, we can stay in  $S$  with an empty stack (and thus accept) no matter what character we see next and no matter what is on the stack (since these transitions don't check the stack). So, if  $M$  accepts all strings in  $\Sigma^*$  of length one, then it accepts all strings in  $\Sigma^*$ . Note that if  $M$  is deterministic, then if it does have all the required transitions it will have no others, since all possible configurations are accounted for

**3. (a)** To determine, given a context-free grammar  $G$ , is  $\epsilon \in L(G)$  This is solvable by using either top down or bottom up parsing on the string  $\epsilon$ .

**(b)** To determine, given a context-free grammar  $G$ , is  $\{\epsilon\} = L(G)$  This is solvable. By the context-free pumping theorem, we know that, given a context-free grammar  $G$  generating a language  $L(G)$ , if there is a string of length greater than  $B^T$  in  $L$ , then  $vy$  can be pumped out to create a shorter string also in  $L$  (the string must be shorter since  $|vy| > 0$ ). We can, of course, repeat this process until we reduce the original string to one of length less than  $B^T$ . This means that if there any strings in  $L$ , there are some strings of length less than  $B^T$ . So, to see whether  $L = \{\epsilon\}$ , we do the following: First see whether  $\epsilon \in L(G)$  by parsing. If not, we say no. If  $\epsilon$  is in  $L$ , then we need to determine whether any other strings are also in  $L$ . To do this, we test all strings in  $\Sigma^*$  of length up to  $B^{T+1}$ . If we find one, we say no,  $L \neq \{\epsilon\}$ . If we don't find any, we can assert that  $L = \{\epsilon\}$ . Why? If there is a longer string in  $L$  and we haven't found it yet, then we know, by the pumping theorem, that we could pump out  $vy$  until we got a string of length  $B^T$  or less. If  $\epsilon$  were not in  $L$ , we could just test up to length  $B^T$  and if we didn't find any elements of  $L$  at all, we could stop, since if there were bigger ones we could pump out and get shorter ones but there aren't any. However, because  $\epsilon$  is in  $L$ , what about the case where we pump out and get  $\epsilon$ ? That's why we go up to  $B^{T+1}$ . If there are any long strings that pump out to  $\epsilon$ , then there is a shortest such string, which can't be longer than  $B^{T+1}$  since that's the longest string we can pump out (by the strong version of the pumping theorem).

**(c)** To determine, given two context-free grammars  $G_1$  and  $G_2$ , is  $L(G_1) \subseteq L(G_2)$  This isn't solvable. If it were, then we could reduce the unsolvable problem of determining whether  $L(G_1) = L(G_2)$  to it. Notice that  $L(G_1) = L(G_2)$  iff  $L(G_1) \subseteq L(G_2)$  and  $L(G_2) \subseteq L(G_1)$ . So, if we could solve the subset problem, then to find out whether  $L(G_1) = L(G_2)$ , all we do is ask whether the first language is a subset of the second and vice versa. If both answers are yes, we say yes. Otherwise, we say no. Formally:

$$L_1 = \{s : s = G_1 G_2, G_1 \text{ and } G_2 \text{ are context-free grammars, and } L(G_1) = L(G_2) \}$$

$$\Downarrow \quad \tau$$

$$(?M_2) \quad L_2 = \{s : s = G_1 G_2, G_1 \text{ and } G_2 \text{ are context-free grammars, and } L(G_1) \subseteq L(G_2) \}$$

If  $M_2$  exists, then  $M_1(G_1 G_2) = M_2(G_1 G_2)$  AND  $M_2(G_2 G_1)$ . To write this out in our usual notation so that the last function that gets applied is  $M_2$ , is sort of tricky, but it can, of course be done: Don't worry about doing it. If you can write any function for  $M_1$  that is guaranteed to be recursive if  $M_2$  exists, then you've done the proof.



**4. (a)** If any language  $L$  is recursively enumerable, then there is a Turing machine that semidecides it. Every Turing machine has a description of finite length. Therefore, the number of Turing machines, and thus the number of recursively enumerable languages, is countably infinite (since the power set of a countable set is countably infinite). If, for some language  $L$ , its complement is re, then it must have a semideciding Turing machine, so there is a countably infinite number of languages whose complement is recursively enumerable. But there is an uncountable number of languages. So there must be languages that are not recursively enumerable and do not have recursively enumerable complements.

**(b)**  $L = \{ \langle M \rangle : M \text{ halts on the input } 0 \text{ and } M \text{ doesn't halt on the input } 1 \}$ .  
The complement of  $L = \{ \langle M \rangle : M \text{ doesn't halt on the input } 0 \text{ or } M \text{ halts on the input } 1 \}$ . Neither of these languages is recursively enumerable because of the doesn't halt piece.

## CS 341 Homework 22 Review

1. Given the following language categories:

- A: L is finite.
- B: L is not finite but is regular.
- C: L is not regular but is deterministic context free
- D: L is not deterministic context free but is context free
- E: L is not context free but is Turing decidable
- F: L is not Turing decidable but is Turing acceptable
- G: L is not Turing acceptable

Assign the appropriate category to each of the following languages. Make sure you can justify your answer.

- a. \_\_\_\_\_  $\{a^n b^{kn} : k = 1 \text{ or } k = 2, n \geq 0\}$
- b. \_\_\_\_\_  $\{a^n b^{kn} : k = 0 \text{ or } k = 1, n \geq 0\}$
- c. \_\_\_\_\_  $\{a^n b^n c^n : n \geq 0\}$
- d. \_\_\_\_\_  $\{a^n b^n c^m : n \geq 0, m \geq 0\}$
- e. \_\_\_\_\_  $\{a^n b^n : n \geq 0\} \cup a^*$
- f. \_\_\_\_\_  $\{a^n b^m : n \text{ is prime and } m \text{ is even}\}$
- g. \_\_\_\_\_  $\{a^n b^m c^{m+n} : n \geq 0, m \geq 0\}$
- h. \_\_\_\_\_  $\{a^n b^m c^{mn} : n \geq 0, m \geq 0\}$
- i. \_\_\_\_\_  $\{a^n b^m : n \geq 0, m \geq 0\}$
- j. \_\_\_\_\_  $\{xy : x \in a^*, y \in b^*, |x| = |y|\}$
- k. \_\_\_\_\_  $\{xy : x \in a^*, y \in a^*, |x| = |y|\}$
- l. \_\_\_\_\_  $\{x : x \in \{a, b, c\}^*, \text{ and } x \text{ has 5 or more a's}\}$
- m. \_\_\_\_\_  $\{ "M" : M \text{ accepts at least 1 string}\}$
- n. \_\_\_\_\_  $\{ "M" : M \text{ is a Turing machine that halts on input } \epsilon \text{ and } | "M" | \leq 1000\}$
- o. \_\_\_\_\_  $\{ "M" : M \text{ is a Turing machine with } \leq 50 \text{ states}\}$
- p. \_\_\_\_\_  $\{ "M" : M \text{ is a Turing machine such that } L(M) = a^*\}$
- q. \_\_\_\_\_  $\{x : x \in \{A, B, C, D, E, F, G\}, \text{ and } x \text{ is the answer you write to this question}\}$

### Solutions

a. D  $\{a^n b^{kn} : k = 1 \text{ or } k = 2, n \geq 0\}$

We haven't discussed many techniques for proving that a context free language isn't deterministic, so we can't prove that this one isn't. But essentially the reason this one isn't is that we don't know what to do when we see b's. Clearly, we can build a pda M to accept this language. As M reads each a, it pushes it onto the stack. When it starts seeing b's, it needs to start popping a's. But there's no way to know, until either it runs out of b's or it gets to the  $(n+1)^{\text{st}}$  b, whether to pop an a for each b or hold back and pop an a for every other b. So M is not deterministic.

b. C  $\{a^n b^{kn} : k = 0 \text{ or } k = 1, n \geq 0\}$

This one looks very similar to **a**, but it's different in one key way. Remember that the definition of deterministic context free is that it is possible to build a deterministic pda to accept  $L\$$ . So now, we can build a deterministic pda M as follows: Push each a onto the stack. When we run out of a's, the next character will either be \$ (in the case where  $k = 0$ ) or b (in the case where  $k = 1$ ). So we know right away which case we're dealing with. If M sees a b, it goes to a state where it pops one b for each a and accepts if it comes out even. If it sees \$, it goes to a state where it clears the stack and accepts.

c. E  $\{a^n b^n c^n : n \geq 0\}$

We proved that this is recursive by showing a grammar for it in Lecture Notes 24. We used the pumping theorem to prove that it isn't context free in Lecture Notes 19.

d. C  $\{a^n b^n c^m : n \geq 0, m \geq 0\}$

This one is context free. We need to compare the a's to the b's, but the c's are independent. So a grammar to generate this one is:

$$\begin{aligned} S &\rightarrow A C \\ A &\rightarrow a A b \\ A &\rightarrow \epsilon \\ C &\rightarrow c C \\ C &\rightarrow \epsilon \end{aligned}$$

It's deterministic because we can build a pda that always knows what to do: push a's, pop an a for each b, then simply scan the c's.

e. C  $\{a^n b^n : n \geq 0\} \cup a^*$

This one is equivalent to **b**, since  $a^* = a^n b^{0n}$ .

f. E  $\{a^n b^m : n \text{ is prime and } m \text{ is even}\}$

This one is recursive because we can write an algorithm to determine whether a number is prime and another one to determine whether a number is even. The proof that it is essentially the same as the one we did in class that  $a^n$ : n is prime is not context free.

g. C  $\{a^n b^m c^{m+n} : n \geq 0, m \geq 0\}$

This one is context free. A grammar for it is:

$$\begin{aligned} S &\rightarrow a S c \\ S &\rightarrow b S c \\ S &\rightarrow \epsilon \end{aligned}$$

It's deterministic because we can build a deterministic pda M for it: M pushes each a onto its stack. It also pushes an a for each b. Then, when it starts seeing c's, it pops one a for each c. If it runs out of a's and c's at the same time, it accepts.

h. E  $\{a^n b^m c^{mm} : n \geq 0, m \geq 0\}$

This one is similar to **g**, but because the number of c's is equal to the product of n and m, rather than the sum, there is no way to know how many c's to generate until we know both how many a's there are and how many b's. Clearly we can write an algorithm to do it, so it's recursive. To prove this, we need to use the pumping theorem. Let  $w = a^M b^M c^{MM}$ . Call the a's region 1, the b's region 2, and the c's region 3. Clearly neither v nor y can span regions since, if they did, we'd get a string with letters out of order. So we need only consider the following possibilities:

- (1, 1) The number of c's will no longer be the product of n and m.
- (1, 2) The number of c's will no longer be the product of n and m.
- (1, 3) Ruled out by  $|vxy| \leq M$ .
- (2, 2) The number of c's will no longer be the product of n and m.
- (2, 3) The number of c's will no longer be the product of n and m.
- (3, 3) The number of c's will no longer be the product of n and m.

i. B  $\{a^n b^m : n \geq 0, m \geq 0\}$

This one is regular. It is defined by the regular expression  $a^*b^*$ . It isn't finite, which we know from the presence of Kleene star in the regular expression.

j. C  $\{xy : x \in a^*, y \in b^*, |x| = |y|\}$

This one is equivalent to  $a^n b^n$ , which we've already shown is context free and not regular. We showed a deterministic pda to accept it in Lecture Notes 14.

k. B  $\{xy : x \in a^*, y \in a^*, |x| = |y|\}$

This one is  $\{w = a^* : |w| \text{ is even}\}$ . We've shown a simple two state FSM for this one.

l. B  $\{x : x \in \{a, b, c\}^*, \text{ and } x \text{ has 5 or more a's}\}$

This one also has a simple FSM F that accepts it. F has six states. It simply counts a's, up to five. If it ever gets to 5, it accepts.

**m. F** {"M" : M accepts at least 1 string}

This one isn't recursive. We know from Rice's Theorem that it can't be, since another way to say this is {"M" : L(M) contains at least 1 string}

We can also show that this one isn't recursive by reduction, which is done in the Supplementary Materials.

**n. A** {"M" : M is a Turing machine that halts on input  $\epsilon$  and  $|M| \leq 1000$ }

This one is finite because of the limit on the length of the strings that can be used to describe M. So it's finite (and thus regular) completely independently of the requirement that M must halt on  $\epsilon$ . You may wonder whether we can actually build a finite state machine F to accept this language. What we know for sure is that F exists. It must for any finite language. Whether we can build it or not is a separate question. The undecidability of the halting problem tells us that we can't build an algorithm to determine whether an arbitrary TM M halts on  $\epsilon$ . But that doesn't mean that we can't look at most Turing Machines and tell. So, here, it is likely that we could write out all the TMs of length less than 1000 and figure out which ones accept  $\epsilon$ . We could then build a deciding FSM F. But even if we can't, that doesn't mean that no such FSM exists. It just means that we don't know what it is. This is no different from the problem of building an FSM to accept all strings of the form mm/dd/yy, such that mm/dd/yy is your birthday. A simple machine F to do this exists. You know how to write it. I don't because I don't know when your birthday is. But that fact that I don't know how to build F says nothing about its existence.

**o. E** {"M" : M is a Turing machine with  $\leq 50$  states}

This one looks somewhat similar to **n**. But it's different in a key way. This set isn't finite because there is no limit on the number of tape symbols that M can use. So we can't do the same trick we can do in **n**, where we could simply list all the machines that met the length restriction. With even a single state, I can build a TM whose description is arbitrarily long. I simply tell it what to do in state one if it's reading character 1. Then what to do if it's reading character 2. Then character 3, and so forth. There's no limit to the number of characters, so there's no limit to the length of the string I must write to consider all of them. Given that the language is not finite, we need a TM to decide it. Why? What we need to do is to check to make sure that the string is a syntactically valid encoding of a Turing Machine. Recall the syntax of an encoding. When we see the first a??? symbol that encodes a tape symbol, we know how many digits it has. All the others must have the same number of digits. So we have to remember that number. Since there's no limit to it, we can't remember it in a finite number of states. Since we need to keep referring to it, we can't remember it on a stack. So we need a TM. But the TM is a straightforward program that will always halt. Thus the language is recursive.

**p. G** {"M" : M is a Turing machine such that  $L(M) = a^*$ }

This one isn't recursive. Again, we know that from Rice's Theorem. And we can prove it by reduction, which we did in the supplementary materials for the more general case of any alphabet  $\Sigma$ . But this language is even harder than many we have considered, such as H. It isn't even recursively enumerable. Why? Informally, the TM languages that are recursive are the ones where we can discover positive instances by simulation (like, H, where we ask whether M halts on a particular w?). But how can we try all strings in  $a^*$ ? Proving this formally is beyond the scope of this class.

**q. A** {x :  $x \in \{A, B, C, D, E, F, G\}$ , and x is the answer you write to this question}

This one is finite. In fact, it is a language of cardinality 1. Thus it's regular and there exists an FSM F that accepts it. You may feel that there's some sort of circularity here. There really isn't, but even if there were, we can use the same argument here that we used in **n**. Even if we didn't know how to build F, we still know that it exists.

# III. Supplementary Materials

# The Three Hour Tour Through Automata Theory

## Analyzing Problems as Opposed to Algorithms

In CS336 you learned to analyze *algorithms*. This semester, we're going to analyze *problems*.

We're going to see that there is a hierarchy of problems: easy, hard, impossible.

For each of the first two, we'll see that for any problem, there are infinitely many programs to solve the problem.

By the way, this is trivial. Take one program and add any number of junk steps. We'll formalize this later once we have some formalisms to work with.

Some may be better than others. But in some cases, we can show some sort of boundary on how good an algorithm we can attempt to find for a particular problem.

## Let's Look at Some Problems

Let's look at a collection of problems, all which could arise in considering one piece of C++ code. [slide - Let's Look at Some Problems]

As we move from problem 1 to problem 5, things get harder in two key ways. The first is that it seems we'll need more complicated, harder to write and design programs. In other words, it's going to take more time to write the programs. The second is that the programs are going to take a lot longer to run. As we get to problems 4 and 5, it's not even clear there is a program that will do the job. In fact, in general for problem 4 there isn't. For problem 5, in general we can't even get a formal statement of the problem, much less an algorithmic solution.

## Languages

### Characterizing Problems as Language Recognition Tasks

In order to create a formal theory of problems (as opposed to algorithms), we need a single, relatively straightforward framework that we can use to describe any kind of possibly computable function. The one we'll use is language recognition.

#### *What is a Language?*

A language is a set of strings over an alphabet, which can be any *finite* collection of symbols. [Slide - Languages]

#### *Defining a Problem as a Language Recognition Task*

We can define any problem as a language recognition task. In other words, we can output just a boolean, True or False. Some problems seem naturally to be described as recognition tasks. For

example, accept grammatical English sentences and reject bad ones. (Although the truth is that English is so squishy it's nearly impossible to formalize this. So let's pick another example -- accept the syntactically valid C programs and reject the others.)

Problems that you think of more naturally as functions can also be described this way. We define the set of input strings to consist of strings that are formed by concatenating an input to an output. Then we only accept strings that have the correct output concatenated to each input.

**[Slide - Encoding Output]**

### *Branching Out -- Allowing for Actual Output*

Although it is simpler to characterize problems simply as recognition problems and it is possible to reformulate functional problems as recognition problems, we will see that we can augment the formalisms we'll develop to allow for output as well.

## **Defining Languages Using Grammars**

Now what we need is a general mechanism for defining languages. Of course, if we have a finite language, we can just enumerate all the strings in it. But most interesting languages are infinite. What we need is a *finite* mechanism for specifying *infinite* languages. Grammars can do this.

The standard way to write a grammar is as a production system, composed of rules with a left hand side and a right hand side. Each side contains a sequence of symbols. Some of these symbols are terminal symbols, i.e., symbols in the language we're defining. Others are drawn from a finite set of nonterminal symbols, which are internal symbols that we use just to help us define the language. Of these nonterminal symbols, one is special -- we'll call it the start symbol.

**[Slide - Grammars 1]**

If there is a grammar that defines a language, then there is an infinite number of such grammars. Some may be better, from various points of view than others. Consider the grammar for odd integers. What different grammars could we write? One thing we could do would be to introduce the idea of odd and even digits. **[Slide - Grammars 2]**

Sometimes we use single characters, disjoint from the characters of the target language, in our rules. But sometimes we need more symbols. Then we often use  $<$  and  $>$  to mark multiple character nonterminal symbols. **[Slide - Grammars 3]**

Notice that we've also introduced a notation for OR so that we don't have to write as many separate rules. By the way, there are lots of ways of writing a grammar of arithmetic expressions. This one is simple but it's not very good. It doesn't help us at all to determine the precedence of operators. Later we'll see other grammars that do that.

## **Grammars as Generators and as Acceptors**

So far, we've defined problems as language recognition tasks. But when you look at the grammars we've considered, you see that there's a sense in which they seem more naturally to be generators than recognizers. If you start with  $S$ , you can generate all the strings in the language

defined by the grammar. We'll see later that we'll use the idea of a grammar as a generator (or an enumerator) as one way to define some interesting classes of languages.

But you can also use grammars as acceptors, as we've suggested. There are two ways to do that. One is *top-down*. By that we mean that you start with S, and apply rules. [**work this out for a simple expression for the Language of Simple Arithmetic Expressions**] At some point, you'll generate a string without any nonterminals (i.e., a string in the language). Check and see if it's the one you want. If so accept. If not, try again. If you do this systematically, then if the string is in the language, you'll eventually generate it. If it isn't, you may or may not know when you should give up. More on that later.

The other approach is bottom up. In this approach, we simply apply the rules sort of backwards, i.e., we run them from right to left, matching the string to the right hand sides of the rules and continuing until we generate S and nothing else. [**work this out for a simple expression for the Language of Simple Arithmetic Expressions**] Again, there are lots of possibilities to consider and there's no guarantee that you'll know when to stop if the string isn't in the language. Actually, for this simple grammar there is, but we can't assure that for all kinds of grammars.

## The Language Hierarchy

Remember that our whole goal in this exercise is to describe classes of problems, characterize them as easy or hard, and define computational mechanisms for solving them. Since we've decided to characterize problems as languages to be recognized, what we need to do is to create a language hierarchy, in which we start with very simple languages and move toward more complex ones.

### *Regular Languages*

Regular languages are very simple languages that can be defined by a very restricted kind of grammar. In these grammars, the left side of every rule is a single nonterminal and the right side is a single terminal optionally followed by a single nonterminal. [**slide - Regular Grammars**] If you look at what's going on with these simple grammars, you can see that as you apply rules, starting with S, you generate a terminal symbol and (optionally) have a new nonterminal to work with. But you can never end up with multiple nonterminals at once. (Recall our first grammar for Odd Integers [**slide - Grammars 1**]).

Of course, we also had another grammar for that same language that didn't satisfy this restriction. But that's okay. If it is possible to define the language using the restricted formalism, then it falls into the restricted class. The fact that there are other, less restricted ways to define it doesn't matter.

It turns out that there is an equivalent, often useful, way to describe this same class of languages, using regular expressions. [**Slide Regular Expressions and Languages**] Regular expressions don't look like grammars, in the sense that there are no production rules, but they can be used to define exactly the same set of languages that the restricted class of regular grammars can define. Here's a regular expression for the language that consists of odd integers, and one for the



language of identifiers. We can try to write a regular expression for the language of matched parenthesis, but we won't succeed.

Intuitively, regular languages are ones that can be defined without keeping track of more than a finite number of things at once. So, looking back at some of our example languages [**slide - Languages**], the first is regular and none of the others is.

### *Context Free Languages*

To get more power in how we define languages, we need to return to the more general production rule structure.

Suppose we allow rules where the left hand side is composed of a single symbol and the right hand side can be anything. We then have the class of context-free grammars. We define the class of context-free languages to include any language that can be generated by a context-free grammar.

The context-free grammar formalism allows us to define many useful languages, including the languages of matched parentheses and of equal numbers of parentheses but in any order [**slide - Context-Free Grammars**]. We can also describe the language of simple arithmetic expressions [**slide - Grammars 3**].

Although this system is a lot more powerful (and useful) than regular languages are, it is not adequate for everything. We'll see some quite simple artificial language it won't work for in a minute. But it's also inadequate for things like ordinary English. [**slide - English Isn't Context-Free**].

### *Recursively Enumerable Languages*

Now suppose we remove all restrictions from the form of our grammars. Any combination of symbols can appear on the left hand side and any combination of symbols can appear on the right. The only real restriction is that there can be only a finite number of rules. For example, we can write a grammar for the language that contains strings of the form  $a^n b^n c^n$ . [**slide - Unrestricted Grammars**]

Once we remove all restrictions, we clearly have the largest set of languages that can be generated by any finite grammar. We'll call the languages that can be generated in this way the class of recursively enumerable languages. This means that, for any recursively enumerable language, it is possible, using the associated grammar, to generate all the strings in the language. Of course, it may take an infinite amount of time if the language contains an infinite number of strings. But any given string, if it is enumerated at all, will be enumerated in a finite amount of time. So I guess we could sit and wait. Unfortunately, of course, we don't know how long to wait, which is a problem if we're trying to decide whether a string is in the language by generating all the strings and seeing if the one we care about shows up.

## *Recursive Languages*

There is one remaining set of languages that it is useful to consider. What about the recursively enumerable languages where we could guarantee that, after a finite amount of time, either a given string would be generated or we would know that it isn't going to be. For example, if we could generate all the strings of length 1, then all the strings of length 2, and so forth, we'd either generate the string we want or we'd just wait until we'd gone past the length we cared about and then report failure. From a practical point of view, this class is very useful since we like to deal with solutions to problems that are guaranteed to halt. We'll call this class of languages the recursive languages. This means that we can not only generate the strings in the language, we can actually, via some algorithm, decide whether a string is in the language and halt, with an answer, either way.

Clearly the class of recursive languages is a subset of the class of recursively enumerable ones. But, unfortunately, this time we're not going to be able to define our new class by placing syntactic restrictions on the form of the grammars we use. There are some useful languages, such as  $a^n b^n c^n$ , that are recursive. There are some others, unfortunately, that are not.

## *The Whole Picture*

[Slide - The Language Hierarchy]

## **Computational Devices**

### **Formal Models of Computational Devices**

If we want to make formal statements about the kinds of computing power required to solve various kinds of problems, then we need simple, precise models of computation.

We're looking for models that make it easy to talk about what can be computed -- we're not worrying about efficiency at this point.

When we described languages and grammars, we saw that we could introduce several different structures, each with different computational power. We can do the same thing with machines. Let's start with really simple devices and see what they can do. When we find limitations, we can expand their power.

### **Finite State Machines**

The only memory consists of the ability to be in one of a finite number of states. The machine operates by reading an input symbol and moving to a new state that is determined solely by the state it is in and the input that it reads. There is a unique start state and one or more final states. If the input is exhausted and the machine is in a final state, then it accepts the input. Otherwise it rejects it.

Example: An FSM to accept odd integers. [Slide - Finite State Machines 1]

Example: An FSM to accept valid identifiers. [Slide - Finite State Machines 2]

Example: How about an FSM to accept strings with balanced parentheses?

Notice one nice feature of every finite state machine -- it will always halt and it will always provide an answer, one way or another. As we'll see later, not all computational systems offer these guarantees.

But we've got this at a price. There is only a finite amount of memory. So, for example, we can't count anything. We need a stronger device.

## Push Down Automata

### *Deterministic PDAs*

Add a single stack to an FSM. Now the action of the machine is a function of its state, the input it reads, and the values at the top of the stack.

Example: A PDA to accept strings with balanced parentheses. [Slide - Push Down Automata]

Notice that this really simple machine only has one state. It's not using the states to remember anything. All its memory is in the stack.

Example: A PDA to accept strings of the form  $w#w^R$ , where  $w \in \{a,b\}^*$  [slide - Pushdown Automaton 2].

Example: How about a PDA to accept strings with some number of a's, followed by the same number of b's, followed by the same number of c's? [slide - PDA 3] It turns out that this isn't possible. The problem is that we could count the a's on the stack, then pop for b's. But how could we tell if there is the right number of c's. We'll see in a bit that we shouldn't be too surprised about this result. We can create PDA's to accept precisely those languages that we can generate with CFGs. And remember that we had to use an unrestricted grammar to generate this language.

### *Nondeterministic PDAs*

Example: How about a PDA to accept strings of the form  $w w^R$ ? We can do this one if we expand our notion of a PDA to allow it to be nondeterministic. The problem is that we don't know when to imagine that the reversal starts. What we need to do is to guess. In particular, we need to try it at every point. We can do this by adding an epsilon transition from the start state (in which we're pushing  $w$ ) to the final state in which we're popping as we read  $w^R$ . [slide - A Nondeterministic PDA] Adding this kind of nondeterminism actually adds power to the PDA notion. And actually, it is the class of nondeterministic PDA's that is equivalent to the class of context-free languages. No surprise, since we were able to write a context free grammar for this language

By the way, it also makes sense to talk about nondeterministic finite state machines. But it turns out that adding nondeterminism to finite state machines doesn't increase the class of things they can compute. It just makes it easier to describe some machines. Intuitively, the reason that nondeterminism doesn't buy you anything with finite state machines is that we can simulate a nondeterministic machine with a deterministic machine. We just make states that represent sets of states in the nondeterministic machine. So in essence, we follow all paths. If one of them accepts, we accept.

Then why can't we do that with PDA's? For finite state machines, there must be a finite number of states. DAH. So there is a finite number of subsets of states and we can just make them the states of our new machine. Clunky but finite. Once we add the stack, however, there is no longer a finite number of states of the total machine. So there is not a finite number of subsets of states. So we can't simulate being in several states at once just using states. And we only have one stack. Which branch would get it? That's why adding nondeterminism actually adds power for PDAs.

## Turing Machines

Clearly there are still some things we cannot do with PDAs. All we have is a single stack. We can count one thing. If we need to count more than one thing (such as a's and b's in the case of languages defined by  $a^n b^n c^n$ ), we're in trouble.

So we need to define a more powerful computing device. The formalism we'll use is called the Turing Machine, after its inventor, Alan Turing. There are many different (and equivalent) ways to write descriptions of Turing Machines, but the basic idea is the same for all of them [**slide - Turing Machines**]. In this new formalism, we allow our machines to write onto the input tape. They can write on top of the input. They can also write past the input. This makes it easier to define computation that actually outputs something besides yes or no if we want to. But, most importantly, because we view the tape as being of infinite length, all limitations of finiteness or limited storage have been removed, even though we continue to retain the core idea of a finite number of states in the controller itself.

Notice, though, that Turing Machines are not guaranteed to halt. Our example one always does. But we could certainly build one that scans right until it finds a blank (writing nothing) and then scans left until it finds the start symbol and then scans right again and so forth. That's a legal (if stupid) Turing Machine. Unfortunately, (see below) it's not always possible to tell, given a Turing Machine, whether it is guaranteed to halt. This is the biggest difference between Turing Machines and the FSMs and PDAs, both of which will always halt.

### *Extensions to Turing Machines*

You may be thinking, wow, this Turing Machine idea sure is restrictive. For example, suppose we want to accept all strings in the simple language  $\{w \# w^R\}$ . We saw that this was easy to do in one pass with a pushdown automaton. But to do this with the sort of Turing Machine we've got so far would be really clunky. [**work this out on a slide**] We'd have to start at the left of the string, mark a character, move all the way to the right to find the corresponding character, mark

it, scan back left, do it again, and so forth. We've just transformed a linear process into an  $n^2$  one.

But suppose we had a Turing Machine with 2 tapes. The first thing we'll do is to copy the input onto the second tape. Now start the read head of the first tape at the left end of the input and the read head of the second tape at the right end. At each step in the operation of the machine, we check to make sure that the characters being read on the two tapes are the same. And we move the head on tape 1 right and the head on tape 2 to the left. We run out of input on both machines at the same time, we accept. **[slide -A Two Head Turing Machine]**

The big question now is, "Have we created a new notational device, one that makes it easier to describe how a machine will operate, or have we actually created a new kind of device with more power than the old one? The answer is the former. We can prove that by showing that we can simulate a Turing Machine with any finite number of tapes by a machine that computes the same thing but only has one tape. **[slide - Simulating k Heads with One]** The key idea here is to use the one tape but to think of it as having some larger number of tracks. Since there is a finite tape alphabet, we know that we can encode any finite number of symbols in a finite (but larger) symbol alphabet. For example, to simulate our two headed machine with a tape alphabet of 3 symbols plus start and blank, we will need  $2*2*5*5$  or 100 tape symbols. So to do this simulation, we must do two main things: Encode all the information from the old, multi-tape machine on the new, single tape machine and redesign the finite state controller so that it simulates, in several moves, each move of the old machine.

It turns out that any "reasonable" addition you can think of to our idea of a Turing Machine is implementable with the simple machine we already have. For example, any nondeterministic Turing Machine can be simulated by a deterministic one. This is really significant. In this, in some ways trivial, machine, we have captured the idea of computability.

Okay, so our Turing Machines can do everything any other machine can do. It also goes the other way. We can propose alternative structures that can do everything our Turing Machines can do. For example, we can simulate any Turing Machine with a deterministic PDA that has two stacks rather than one. What this machine will do is read its input tape once, copying onto the first stack all the nonblank symbols. Then it will pop all those symbols off, one at a time, and move them to the second stack. Now it can move along its simulated tape by transferring symbols from one stack to the other. **[slide - Simulating a Turing Machine with Two Stacks]**

### *The Universal Turing Machine*

So now, having shown that we can simulate anything on a simple Turing Machine, it should come as no surprise that we can design a Turing Machine that takes as its input the definition of another Turing Machine, along with an input for that machine. What our machine does is to simulate the behavior of the machine it is given, on the given input.

Remember that to simulate a k-tape machine by a 1 tape machine we had first to state how to encode the multiple tapes. Then we had to state how the machine would operate on the encoding. We have to do the same thing here. First we need to decide how to encode the states

and the tape symbols of the input machine, which we'll call  $M$ . There's no upper bound on how many states or tape symbols there will be. So we can't encode them with single symbols. Instead we'll encode states as strings that start with a "q" and then have a binary encoding of the state number (with enough leading zeros so all such encodings take the same number of digits). We'll encode tape symbols as an "a" followed by a binary encoding of the count of the symbol. And we'll encode "move left" as 10, "move right" as 01, and stay put as 00. We'll use # as a delimiter between transitions. [slide - Encoding States, Symbols, and Transitions]

Next, we need a way to encode the simulation of the operation of  $M$ . We'll use a three tape machine as our Universal Turing Machine. (Remember, we can always implement it on a one tape machine, but this is a lot easier to describe.) We'll use one tape to encode the tape of  $M$ , the second tape contains the encoding of  $M$ , and the third tape encodes the current state of  $M$  during the simulation. [slide - The Universal Turing Machine]

### **A Hierarchy of Computational Devices**

These various machines that we have just defined, fall into an inclusion hierarchy, in the sense that the simpler machines can always be simulated by the more powerful ones. [Slide - A Machine Hierarchy]

### **The Equivalence of the Language Hierarchy and the Computational Hierarchy**

Okay, this probably comes as no surprise. The machine hierarchy we've just examined exactly mirrors the language hierarchy. [Slide - Languages and Machines]

Actually, this is an amazing result. It seems to suggest that there's something quite natural about these categories.

### **Church's Thesis**

If we really want to talk about naturalness, can we say anything about whether we've captured what it means to be computable? Church's Thesis (also sometimes called the Church-Turing Thesis) asserts that the precise concept of the Turing Machine that halts on all inputs corresponds to the intuitive notion of an algorithm. Think about it. Clearly a Turing Machine that halts defines an algorithm. But what about the other way around? Could there be something that is computable by some kind of algorithm that is not computable by a Turing Machine that halts? From what we've seen so far, it may seem unlikely, since every extension we can propose to the Turing Machine model turns out possibly to make things more convenient, but it never extends the formal power. It turns out that people have proposed various other formalisms over the last 50 years or so, and they also turn out to be no more powerful than the Turing Machine. Of course, something could turn up, but it seems unlikely.

## **Techniques for Showing that a Problem (or Language) Is Not in a Particular Circle in the Hierarchy**

### **Counting**

$a^n b^n$  is not regular.

### **Closure Properties**

$L = \{ a^n b^m c^p : m \neq n \text{ or } m \neq p \}$  is not deterministic context-free. [slide - Using Closure Properties] Notice that  $L'$  contains all strings that violate at least one of the requirements for  $L$ . So they may be strings that aren't composed of a string of a's, followed by a string of b's, followed by a string of c's. Or they may have that property but they violate the rule that  $m$ ,  $n$ , and  $p$  cannot all be the same. In other words, they are all the same. So if we intersect  $L'$  with the regular expression  $a^*b^*c^*$ , we throw away everything that isn't a string of a's then b's then c's, and we're left with strings of  $n$  a's, followed, by  $n$  b's, followed by  $n$  c's.

### **Diagonalization**

Remember the proof that the power set of the integers isn't countable. If it were, there would be a way of enumerating the sets, thus setting them in one to one correspondence with the integers. But suppose there is such a way [slide - Diagonalization]. Then we could represent it in a table where element  $(i, j)$  is 1 precisely in case the number  $j$  is present in the set  $i$ . But now construct a new set, represented as a new row of the table. In this new row, element  $i$  will be 1 if element  $(i, i)$  of the original table was 0, and vice versa. This row represents a new set that couldn't have been in the previous enumeration. Thus we get a contradiction and the power set of the integers must not be countable.

We can use this technique for perhaps the most important result in the theory of computing.

## **The Unsolvability of the Halting Problem**

There are recursively enumerable languages that are not recursive. In other words, there are sets that can be enumerated, but there is no decision procedure for them. Any program that attempts to decide whether a string is in the language may not halt.

One of the most interesting such sets is the following. Consider sets of ordered pairs where the first element is a description of a Turing Machine. The second element is an input to the machine. We want to include only those ordered pairs where the machine halts on the input. This set is not recursively enumerable. In other words, there's no way to write an algorithm that, given a machine and an input, determines whether or not the machine halts on the input. [slide - The Unsolvability of the Halting Problem]

Suppose there were such a machine. Let's call it HALTS.  $\text{HALTS}(M, x)$  returns true if Turing machine  $M$  halts on input  $x$ . Otherwise it returns false. Now we write a Turing Machine program that implements the TROUBLE algorithm. Now what happens if we invoke  $\text{HALTS}(\text{TROUBLE}, \text{TROUBLE})$ ? If HALTS says true, namely that TROUBLE will halt on

itself, then TROUBLE loops (i.e., it doesn't halt, thus contradicting our assumption that HALTS could do the job). But if HALTS says FALSE, namely that TROUBLE will not halt on itself, then TROUBLE promptly halts, thus again proving our supposed oracle HALTS wrong. Thus HALTS cannot exist.

We've used a sort of stripped down version of diagonalization here [**slide - Viewing the Halting Problem as Diagonalization**] in which we don't care about the whole row of the item that creates the contradiction. We're only invoking HALTS with two identical inputs. It's just the single element that we care about and that causes the problem.

### **Let's Revisit Some Problems**

Let's look again at the collection of problems that we started this whole process with. [**slide - Let's Revisit Some Problems**]

Problem 1 can be solved with a finite state machine.

Problem 2 can be solved with a PDA.

Problem 3 can be solved with a Turing Machine.

Problem 4 can be semi solved with a Turing Machine, but it isn't guaranteed to halt.

Problem 5 can't even be stated.

### **So What's Left?**



# Review of Mathematical Concepts

## 1 Sets

### 1.1 What is a Set?

A *set* is simply a collection of objects. The objects (which we call the *elements* or *members* of the set) can be anything: numbers, people, fruits, whatever. For example, all of the following are sets:

$A = \{13, 11, 8, 23\}$

$B = \{8, 23, 11, 13\}$

$C = \{8, 8, 23, 23, 11, 11, 13, 13\}$

$D = \{\text{apple, pear, banana, grape}\}$

$E = \{\text{January, February, March, April, May, June, July, August, September, October, November, December}\}$

$F = \{x : x \in E \text{ and } x \text{ has 31 days}\}$

$G = \{\text{January, March, May, July, August, October, December}\}$

$N = \text{the nonnegative integers}$  (We will generally call this set  $N$ , the *natural numbers*.)

$H = \{i : \exists x \in N \text{ and } i = 2x\}$

$I = \{0, 2, 4, 6, 8, \dots\}$

$J = \text{the even natural numbers}$

$K = \text{the syntactically valid C programs}$

$L = \{x : x \in K \text{ and } x \text{ never gets into an infinite loop}\}$

$Z = \text{the integers}$  ( $\dots -3, -2, -1, 0, 1, 2, 3, \dots$ )

In the definitions of  $F$  and  $H$ , we have used the colon notation. Read it as "such that". We've also used the standard symbol  $\in$  for "element of". We will also use  $\notin$  for "not an element of". So, for example,  $17 \notin A$  is true.

Remember that a set is simply a collection of elements. So if two sets contain precisely the same elements (regardless of the way we actually defined the sets), then they are identical. Thus  $F$  and  $G$  are the same set, as are  $H$ ,  $I$ , and  $J$ .

An important aside: **Zero is an even number.** This falls out of any reasonable definition we can give for even numbers. For example, the one we used to define set  $H$  above. Or consider: 2 is even and any number that can be derived by adding or subtracting 2 from an even number is also even. In order to construct a definition for even numbers that does not include zero, we'd have to make a special case. That would make for an inelegant definition, which we hate. And, as we'll see down the road, we'd also have to make corresponding special cases for zero in a wide variety of algorithms.

Since a set is defined only by what elements it contains, it does not matter what order we list the elements in. Thus  $A$  and  $B$  are the same set.

Our definition of a set considers only whether or not an element is contained within the set. It does not consider how many times the element is mentioned. In other words, duplicates don't count. So  $A$ ,  $B$ , and  $C$  are all equal.

Whenever we define a set, it would be useful if we could also specify a decision procedure for it. A *decision procedure* for a set  $S$  is an algorithm that, when presented with an object  $O$ , returns True if  $O \in S$  and False otherwise. Consider set  $K$  above (the set of all syntactically valid C programs). We can easily decide whether or not an object is an element of  $K$ . First, of course, it has to be a string. If you bring me an apple, I immediately say no. If it is a string, then I can feed it to a C compiler and let it tell me whether or not the object is in  $K$ . But now consider the set  $L$  (C programs that are guaranteed to halt on all inputs). Again, I can reject apples and anything else that isn't even in  $K$ . I can also reject some programs that clearly do loop forever. And I can accept some C programs, for example ones that don't contain any loops at all. But what about the general problem. Can I find a way to look at an arbitrary C program and tell whether or not it belongs in  $L$ . It turns out, as we'll see later, that the answer to this is no. We can prove that no program to solve this problem can exist. But that doesn't mean that the set  $L$  doesn't exist. It's a perfectly fine set. There just isn't a decision procedure for it.

The smallest set is the set that contains no elements. It is called the **empty set**, and is written  $\emptyset$  or  $\{\}$ .

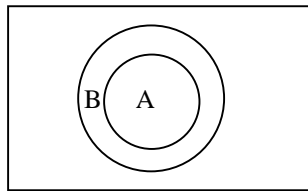
When you are working with sets, it is very important to keep in mind the difference between a set and the elements of a set. Given a set that contains more than one element, this is not usually tricky. It's clear that  $\{1, 2\}$  is distinct from either the number 1 or the number 2. It sometimes becomes a bit trickier though with **singleton sets** (sets that contain only a single element). But it is equally true here. So, for example,  $\{1\}$  is distinct from the number 1. As another example, consider  $\{\emptyset\}$ . This is a set that contains one element. That element is in turn a set that contains no elements (i.e., the empty set).

## 1.2 Relating Sets to Each Other

We say that A is a **subset** of B (which we write as  $A \subseteq B$ ) if every element of A is also an element of B. The symbol we use for subset ( $\subseteq$ ) looks somewhat like  $\leq$ . This is no accident. If  $A \subseteq B$ , then there is a sense in which the set A is "less than or equal to" the set B, since all the elements of A must be in B, but there may be elements of B that are not in A.

Given this definition, notice that every set is a subset of itself. This fact turns out to offer us a useful way to prove that two sets A and B are equal: First prove that A is a subset of B. Then prove that B is a subset of A. We'll have more to say about this later in Section 6.2.

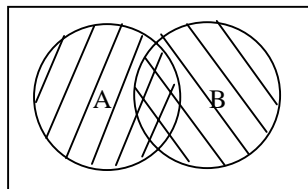
We say that A is a **proper subset** of B (written  $A \subset B$ ) if  $A \subseteq B$  and  $A \neq B$ . The following Venn diagram illustrates the proper subset relationship between A and B:



Notice that the empty set is a subset of every set (since, trivially, every element of  $\emptyset$ , all none of them, is also an element of every other set). And the empty set is a **proper** subset of every set other than itself.

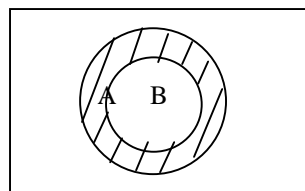
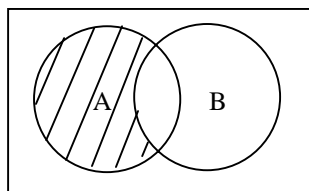
It is useful to define some basic operations that can be performed on sets:

The **union** of two sets A and B (written  $A \cup B$ ) contains all elements that are contained in A or B (or both). We can easily visualize union using a Venn diagram. The union of sets A and B is the entire hatched area:

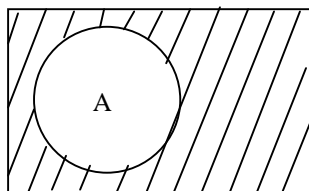


The **intersection** of two sets A and B (written  $A \cap B$ ) contains all elements that are contained in both A and B. In the Venn diagram shown above, the intersection of A and B is the double hatched area in the middle.

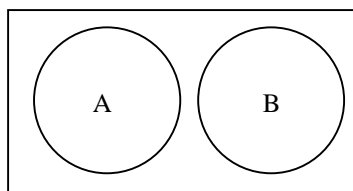
The **difference** of two sets A and B (written  $A - B$ ) contains all elements that are contained in A but not in B. In both of the following Venn diagrams, the hatched region represents  $A - B$ .



The **complement** of a set  $A$  with respect to a specific domain  $D$  (written as  $\bar{A}$  or  $\neg A$ ) contains all elements of  $D$  that are not contained in  $A$  (i.e.,  $\bar{A} = D - A$ ). For example, if  $D$  is the set of residents of Austin and  $A$  is the set of Austin residents who like barbeque, then  $\bar{A}$  is the set of Austin residents who don't like barbeque. The complement of  $A$  is shown as the hatched region of the following Venn diagram:



Two sets are **disjoint** if they have no elements in common (i.e., their intersection is empty). In the following Venn diagram,  $A$  and  $B$  are disjoint:



So far, we've talked about operations on pairs of sets. But just as we can extend binary addition and sum up a whole set of numbers, we can extend the binary operations on sets and perform them on sets of sets. Recall that for summation, we have the notation

$$\sum A_i$$

Similarly, we'll introduce

$$\bigcup A_i \quad \text{and} \quad \bigcap A_i$$

to indicate the union of a set of sets and the intersection of a set of sets, respectively.

Now consider a set  $A$ . For example, let  $A = \{1, 2, 3\}$ . Next, let's enumerate the set of all subsets of  $A$ :

$$\{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

We call this set the **power set** of  $A$ , and we write it  $2^A$ . The power set of  $A$  is interesting because, if we're working with the elements of  $A$ , we may well care about all the ways in which we can combine those elements.

Now for one final property of sets. Again consider the set  $A$  above. But this time, rather than looking for all possible subsets, let's just look for a single way to carve  $A$  up into subsets such that each element of  $A$  is in precisely one subset. For example, we might choose any of the following sets of subsets:

$$\{\{1\}, \{2, 3\}\} \quad \text{or} \quad \{\{1, 3\}, \{2\}\} \quad \text{or} \quad \{\{1, 2, 3\}\}$$

We call any such set of subsets a **partition** of  $A$ . Partitions are very useful. For example, suppose we have a set  $S$  of students in a school. We need for every student to be assigned to precisely one lunch period. Thus we must construct a partition of  $S$ : a set of subsets, one for each lunch period, such that each student is in precisely one subset. More formally, we say that  $\Pi$  is a **partition** of a set  $A$  if and only if (a) no element of  $\Pi$  is empty; (b) all members of  $\Pi$  are disjoint (alternatively, each element of  $A$  is in only one element of  $\Pi$ ); and (c)  $\bigcup \Pi = A$  (alternatively, each element of  $A$  is in some element of  $\Pi$  and no element not in  $A$  is in any element of  $\Pi$ ).

This notion of partitioning a set is fundamental to programming. Every time you analyze the set of possible inputs to your program and consider the various cases that must be dealt with, you're forming a partition of the set of inputs: each input must fall through precisely one path in your program. So it should come as no surprise that, as we build formal models of computational devices, we'll rely heavily on the idea of a partition on a set of inputs as an analytical technique.

## 2 Relations and Functions

In the last section, we introduced some simple relations that can hold between sets (subset and proper subset) and we defined some operations (functions) on sets (union, intersection, difference, and complement). But we haven't yet defined formally what we mean by a relation or a function. Let's do that now. (By the way, the reason we introduced relations and functions on sets in the last section is that we're going to use sets as the basis for our formal definitions of relations and functions and we will need the simple operations we just described as part of our definitions.)

### 2.1 Relations

An *ordered pair* is a sequence of two objects. Given any two objects,  $x$  and  $y$ , there are two ordered pairs that can be formed. We write them as  $(x, y)$  and  $(y, x)$ . As the name implies, in an ordered pair (as opposed to in a set), order matters (unless  $x$  and  $y$  happen to be equal).

The *Cartesian product* of two sets  $A$  and  $B$  (written  $A \times B$ ) is the set of all ordered pairs  $(a, b)$  such that  $a \in A$  and  $b \in B$ . For example, let  $A$  be a set of people {Dave, Sue, Billy} and let  $B$  be a set of desserts {cake, pie, ice cream}. Then

$$A \times B = \{ (Dave, cake), (Dave, pie), (Dave, ice cream), \\ (Sue, cake), (Sue, pie), (Sue, ice cream), \\ (Billy, cake), (Billy, pie), (Billy, ice cream) \}$$

As you can see from this example, the Cartesian product of two sets contains elements that represent all the ways of pairing someone from the first set with someone from the second. Note that  $A \times B$  is not the same as  $B \times A$ . In our example,

$$B \times A = \{ (cake, Dave), (pie, Dave), (ice cream, Dave), \\ (cake, Sue), (pie, Sue), (ice cream, Sue), \\ (cake, Billy), (pie, Billy), (ice cream, Billy) \}$$

We'll have more to say about the cardinality (size) of sets later, but for now, let's make one simple observation about the cardinality of a Cartesian product. If  $A$  and  $B$  are finite and if there are  $p$  elements in  $A$  and  $q$  elements in  $B$ , then there are  $p \cdot q$  elements in  $A \times B$  (and in  $B \times A$ ).

We're going to use Cartesian product a lot. It's our basic tool for constructing complex objects out of simpler ones. For example, we're going to define the class of Finite State Machines as the Cartesian product of five sets. Each individual finite state machine then will be a five tuple  $(K, \Sigma, \delta, s, F)$  drawn from that Cartesian product. The sets will be:

1. The set of all possible sets of states:  $\{\{q1\}, \{q1, q2\}, \{q1, q2, q3\}, \dots\}$ . We must draw  $K$  from this set.
2. The set of all possible input alphabets:  $\{\{a\}, \{a, b, c\}, \{\alpha, \beta, \gamma\}, \{1, 2, 3, 4\}, \{1, w, h, j, k\}, \{q, a, f\}, \{a, \beta, 3, j, f\} \dots\}$ . We must draw  $\Sigma$  from this set.
3. The set of all possible transition functions, which tell us how to move from one state to the next. We must draw  $\delta$  from this set.
4. The set of all possible start states. We must draw  $s$  from this set.
5. The set of all possible sets of final states. (If we land in one of these when we've finished processing an input string, then we accept the string, otherwise we reject.) We must draw  $F$  from this set.

Let's return now to the simpler problem of choosing dessert. Suppose we want to define a relation that tells us, for each person, what desserts he or she likes. We might write the Dessert relation, for example as

$$\{(Dave, cake), (Dave, ice cream), (Sue, pie), (Sue, ice cream)\}$$

In other words, Dave likes cake and ice cream, Sue likes pie and ice cream, and Billy hates desserts.

We can now define formally what a relation is. A *binary relation* over two sets  $A$  and  $B$  is a subset of  $A \times B$ . Our dessert relation clearly satisfies this definition. So do lots of other relations, including common ones defined on the integers. For

example, Less than (written  $<$ ) is a binary relation on the integers. It contains an infinite number of elements drawn from the Cartesian product of the set of integers with itself. It includes, for example:

$$\{(1,2), (2,3), (3,4), \dots\}$$

Notice several important properties of relations as we have defined them. First, a relation may be equal to the empty set. For example, if Dave, Sue, and Billy all hate dessert, then the dessert relation would be  $\{\}$  or  $\emptyset$ .

Second, there are no constraints on how many times a particular element of A or B may occur in the relation. In the dessert example, Dave occurs twice, Sue occurs twice, Billy doesn't occur at all, cake occurs once, pie occurs once, and ice cream occurs twice.

If we have two or more binary relations, we may be able combine them via an operation we'll call composition. For example, if we knew the number of fat grams in a serving of each kind of dessert, we could ask for the number of fat grams in a particular person's dessert choices. To compute this, we first use the Dessert relation to find all the desserts each person likes. Next we get the bad news from the FatGrams relation, which probably looks something like this:

$$\{(cake, 25), (pie, 15), (ice\ cream, 20)\}$$

Finally, we see that the composed relation that relates people to fat grams is  $\{(Dave, 25), (Dave, 20), (Sue, 15), (Sue, 20)\}$ . Of course, this only worked because when we applied the first relation, we got back desserts, and our second relation has desserts as its first component. We couldn't have composed Dessert with Less than, for example.

Formally, we say that the **composition** of two relations  $R_1 \subseteq A \times B$  and  $R_2 \subseteq B \times C$ , written  $R_2 \circ R_1$  is  $\{(a,c) : \exists (a,b) \in R_1 \text{ and } (b,c) \in R_2\}$ . Note that in this definition, we've said that to compute  $R_2 \circ R_1$ , we first apply  $R_1$ , then  $R_2$ . In other words we go right to left. Some definitions go the other way. Obviously we can define it either way, but it's important to check carefully what definition people are using and to be consistent in what you do. Using this notation, we'd represent the people to fat grams composition described above as  $FatGrams \circ Dessert$ .

Now let's generalize a bit. An ordered pair is a sequence (where order counts) of two elements. We could also define an ordered triple as a sequence of three elements, an ordered quadruple as a sequence of four elements, and so forth. More generally, if n is any positive integer, then an **ordered n-tuple** is a sequence of n elements. For example, (Ann, Joe, Mark) is a 3-tuple.

We defined binary relation using our definition of an ordered pair. Now that we've extended our definition of an ordered pair to an ordered n-tuple, we can extend our notion of a relation to allow for an arbitrary number of elements to be related. We define an **n-ary relation** over sets  $A_1, A_2, \dots, A_n$  as a subset of  $A_1 \times A_2 \times \dots \times A_n$ . The n sets may be different, or they may be the same. For example, let A be a set of people:

$$A = \{Dave, Sue, Billy, Ann, Joe, Mark, Cathy, Pete\}$$

Now suppose that Ann and Dave are the parents of Billy, Ann and Joe are the parents of Mark, and Mark and Sue are the parents of Cathy. Then we could define a 3-ary (or **ternary**) relation Child-of as the following subset of  $A \times A \times A$ :

$$\{(Ann, Dave, Billy), (Ann, Joe, Mark), (Mark, Sue, Cathy)\}$$

## 2.2 Functions

Relations are very general. They allow an object to be related to any number of other objects at the same time (as we did in the dessert example above). Sometimes, we want a more restricted notion, in which each object is related to a unique other object. For example, (at least in an ideal world without criminals or incompetent bureaucrats) each American resident is related to a unique social security number. To capture this idea we need functions. A **function** from a set A to a set B is a special kind of a binary relation over A and B in which each element of A occurs precisely once. The dessert relation we defined earlier is not a function since Dave and Sue each occur twice and Billy doesn't occur at all. We haven't restricted each person to precisely one dessert. A simple relation that *is* a function is the successor function Succ defined on the integers:

$$Succ(n) = n + 1.$$

Of course, we cannot write out all the elements of Succ (since there are an infinite number of them), but Succ includes:

$$\{\dots, (-3, -2), (-2, -1), (-1, 0), (0, 1), (1, 2), (2, 3), \dots\}$$

It's useful to define some additional terms to make it easy to talk about functions. We start by writing

$$f: A \rightarrow B,$$

which means that  $f$  is a function from the set  $A$  to the set  $B$ . We call  $A$  the *domain* of  $f$  and  $B$  the *codomain* or *range*. We may also say that  $f$  is a function from  $A$  to  $B$ . If  $a \in A$ , then we write

$$f(a),$$

which we read as "f of a" to indicate the element of  $B$  to which  $a$  is related. We call this element the *image* of  $a$  under  $f$  or the *value* of  $f$  for  $a$ . Note that, given our definition of a function, there must be exactly one such element. We'll also call a the *argument* of  $f$ . For example we have that

$$\text{Succ}(1) = 2, \text{ Succ}(2) = 3, \text{ and so forth.}$$

Thus 2 is the image (or the value) of the argument 1 under Succ.

Succ is a *unary function*. It maps from a single element (a number) to another number. But there are lots of interesting functions that map from ordered pairs of elements to a value. We call such functions *binary functions*. For example, integer addition is a binary function:

$$+: (Z \times Z) \rightarrow Z$$

Thus  $+$  includes elements such as  $((2, 3), 5)$ , since  $2 + 3$  is 5. We could also write

$$+((2,3)) = 5$$

We have double parentheses here because we're using the outer set to indicate function application (as we did above without confusion for Succ) and the inner set to define the ordered pair to which the function is being applied. But this is confusing. So, generally, when the domain of a function is the Cartesian product of two or more sets, as it is here, we drop the inner set of parentheses and simply write

$$+(2,3) = 5.$$

Alternatively, many common binary functions are written in infix notation rather than the prefix notation that is standard for all kinds of function. This allows us to write

$$2+3 = 5$$

So far, we've had unary functions and binary functions. But just as we could define n-ary relations for arbitrary values of  $n$ , we can define n-ary functions. For any positive integer  $n$ , an *n-ary function*  $f$  is a function is defined as

$$F: (D_1 \times D_2 \dots \times D_n) \rightarrow R$$

For example, let  $Z$  be the set of integers. Then

$$\text{QuadraticEquation}: (Z \times Z \times Z) \rightarrow F$$

is a function whose domain is an ordered triple of integers and whose domain is a set of functions. The definition of Quadratic Equation is:

$$\text{QuadraticEquation}(a, b, c)(x) = ax^2 + bx + c$$

What we did here is typical of function definition. First we specify the domain and the range of the function. Then we define how the function is to compute its value (an element of the range) given its arguments (an element of the domain). QuadraticEquation may seem a bit unusual since its range is a set of functions, but both the domain and the range of a function can be any set of objects, so sets of functions qualify.

Recall that in the last section we said that we could compose binary relations to derive new relations. Clearly, since functions are just special kinds of binary relations, if we can compose binary relations we can certainly compose binary functions. Because a function returns a unique value for each argument, it generally makes a lot more sense to compose functions than it does relations, and you'll see that although we rarely compose relations that aren't functions, we compose functions all the time. So, following our definition above for relations, we define the *composition of two functions*  $F_1 \subseteq A \times B$  and  $F_2 \subseteq B \times C$ , written  $F_2 \circ F_1$  is  $\{(a,c) : \exists b (a, b) \in F_1 \text{ and } (b, c) \in F_2\}$ . Notice that the composition of two functions must necessarily also be a function. We mentioned above that there is sometimes confusion about the order in which relations (and now functions) should be applied when they are composed. To avoid this problem, let's introduce a new notation  $F(G(x))$ . We use the parentheses here to indicate function application, just as we did above. So this notation is clear. Apply  $F$  to the result of first applying  $G$  to  $x$ . This notation reads right to left as does our definition of the  $\circ$  notation.

A function is a special kind of a relation (one in which each element of the domain occurs precisely once). There are also special kinds of functions:

A function  $f : D \rightarrow R$  is **total** if it is defined for every element of  $D$  (i.e., every element of  $D$  is related to some element of  $R$ ). The standard mathematical definition of a function requires totality. The reason we haven't done that here is that, as we pursue the idea of "computable functions", we'll see that there are total functions whose domains cannot be effectively defined (for example, the set of C programs that always halt). Thus it is useful to expand the definition of the function's domain (e.g., to the set of all C programs) and acknowledge that if the function is applied to certain elements of the domain (e.g., programs that don't halt), its value will be undefined. We call this broader class of functions (which does include the total functions as a subset) the set of **partial** functions. For the rest of our discussion in this introductory unit, we will consider only total functions, but be prepared for the introduction of partial functions later.

A function  $f : D \rightarrow R$  is **one to one** if no element of the range occurs more than once. In other words, no two elements of the domain map to the same element of the range. Succ is one to one. For example, the only number to which we can apply Succ and derive 2 is 1. QuadraticEquation is also one to one. But + isn't. For example, both  $+(2,3)$  and  $+(4,1)$  equal 5.

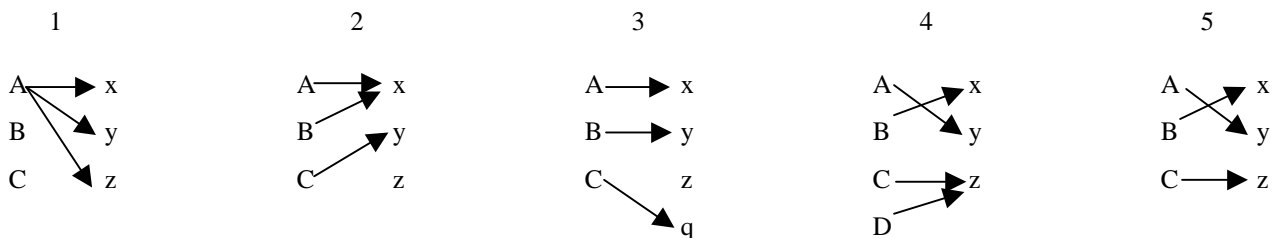
A function  $f : D \rightarrow R$  is **onto** if every element of  $R$  is the value of some element of  $D$ . Another way to think of this is that a function is onto if all of the elements of the range are "covered" by the function. As we defined it above, Succ is onto. But let's define a different function Succ' on the natural numbers (rather than the integers). So we define

$$\text{Succ}' : \mathbb{N} \rightarrow \mathbb{N}.$$

Succ' is not onto because there is no natural number  $i$  such that  $\text{Succ}'(i) = 0$ .

The easiest way to envision the differences between an arbitrary relation, a function, a one to one function and an onto function is to make two columns (the first for the domain and the second for the range) and think about the sort of matching problems you probably had on tests in elementary school.

Let's consider the following five matching problems and let's look at various ways of relating the elements of column 1 (the domain) to the elements of column 2 (the range):



The relationship in example 1 is a relation but it is not a function, since there are three values associated A. The second example is a function since, for each object in the first column, there is a single value in the second column. But this function is neither one to one (because  $x$  is derived from both  $A$  and  $B$ ) nor onto (because  $z$  can't be derived from anything). The third example is a function that is one to one (because no element of the second column is related to more than one element of the first column). But it still isn't onto because  $z$  has been skipped: nothing in the first column derives it. The fourth example is a function that is onto (since every element of column two has an arrow coming into it), but it isn't one to one, since  $z$  is derived from both  $C$  and  $D$ . The fifth and final example is a function that is both one to one and onto. By the way, see if you can modify either example 3 or example 4 to make them both one to one and onto. You're not allowed to change the number of elements in either column, just the arrows. You'll notice that you can't do it. In order for a function to be both one to one and onto, there must be equal numbers of elements in the domain and the range.

The **inverse** of a binary relation  $R$  is simply the set of ordered pairs in  $R$  with the elements of each pair reversed. Formally, if  $R \subseteq A \times B$ , then  $R^{-1} \subseteq B \times A = \{(b, a) : (a, b) \in R\}$ . If a relation is a way of associating with each element of  $A$  a corresponding element of  $B$ , then think of its inverse as a way of associating with elements of  $B$  their corresponding elements in  $A$ . Every relation has an inverse. Every function also has an inverse, but that inverse may not also be a function. For example, look again at example two of the matching problems above. Although it is a function, its inverse is not. Given the argument  $x$ , should we return the value  $A$  or  $B$ ? Now consider example 3. Its inverse is also not a (total) function, since there is no value to be returned for the argument  $z$ . Example four has the same problem example

two does. Now look at example five. Its inverse is a function. Whenever a function is both one to one and onto, its inverse will also be a function and that function will be both one to one and onto.

Inverses are useful. When a function has an inverse, it means that we can move back and forth between columns one and two without loss of information. Look again at example five. We can think of ourselves as operating in the  $\{A, B, C\}$  universe or in the  $\{x, y, z\}$  universe interchangeably since we have a well defined way to move from one to the other. And if we move from column one to column two and then back, we'll be exactly where we started. Functions with inverses (alternatively, functions that are both one to one and onto) are called *bijections*. And they may be used to define *isomorphisms* between sets, i.e., formal correspondences between the elements of two sets, often with the additional requirement that some key structure be preserved. We'll use this idea a lot. For example, there exists an isomorphism between the set of states of a finite state machine and a particular set of sets of input strings that could be fed to the machine. In this isomorphism, each state is associated with precisely the set of strings that drive the machine to that state.

### 3 Binary Relations on a Single Set

Although it makes sense to talk about n-ary relations, for arbitrary values of n (and we have), it turns out that the most useful relations are often binary ones -- ones where n is two. In fact, we can make a further claim about some of the most useful relations we'll work with: they involve just a single set. So instead of being subsets of  $A \times B$ , for arbitrary values of A and B, they are subsets of  $A \times A$ , for some particular set of A of interest. So let's spend some additional time looking at this restricted class of relations.

#### 3.1 Representing Binary Relations on a Single Set

If we're going to work with some binary relation R, and, in particular, if we are going to compute with it, we need some way to represent the relation. We have several choices. We could:

- 1) List the elements of R.
- 2) Encode R as a computational procedure. There are at least two ways in which a computational procedure can define R. It may:
  - a) enumerate the elements of R, or
  - b) return a boolean value, when given an ordered pair. True means that the pair is in R; False means it is not.
- 3) Encode R as a directed graph.
- 4) Encode R as an *incidence matrix*.

For example, consider the mother-of relation M in a family in which Doreen is the mother of Ann, Ann is the mother of Catherine, and Catherine is the mother of Allison. To exploit approach 1, we just write

$$M = \{(Doreen, Ann), (Ann, Catherine), (Catherine, Allison)\}.$$

Clearly, this approach only works for finite relations.

The second approach simply requires code appropriate to the particular relation we're dealing with. One appeal of this approach is that it works for both finite and infinite relations, although, of course, a program that enumerates elements of an infinite relation will never halt.

Next we consider approach 3. Assuming that we are working with a finite relation  $R \subseteq A \times A$ , we can build a directed graph to represent R as follows:

- 1) Construct a set of nodes, one for each element of A that appears in any element of R.
- 2) For each ordered pair in R, draw an edge from the first element of the pair to the second.

The following directed graph represents our example relation M defined above:





And, finally, approach 4: Again assuming a finite relation  $R \subseteq A \times A$ , we can build an incidence matrix to represent  $R$  as follows:

- 1) Construct a square boolean matrix  $S$  whose number of rows and columns equals the number of elements of  $A$  that appear in any element of  $R$ .
- 2) Label one row and one column for each such element of  $A$ .
- 3) For each element  $(p, q)$  of  $R$ , set  $S(p, q)$  to 1 (or True). Set all other elements of  $S$  to 0 (or False).

The following boolean matrix represents our example relation  $M$  defined above:

	Doreen	Ann	Catherine	Allison
Doreen	0	1	0	0
Ann	0	0	1	0
Catherine	0	0	0	1
Allison	0	0	0	0

### 3.2 Properties of Relations

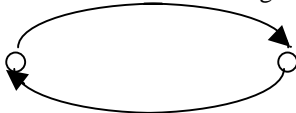
Many useful binary relations have some kind of structure. For example, it might be the case that every element of the underlying set is related to itself. Or it might happen that if  $a$  is related to  $b$ , then  $b$  must necessarily be related to  $a$ . There's one special kind of relation, called an equivalence relation that is particularly useful. But before we can define it, we need first to define each of the individual properties that equivalence relations possess.

A relation  $R \subseteq A \times A$  is **reflexive** if, for each  $a \in A$ ,  $(a, a) \in R$ . In other words a relation  $R$  on the set  $A$  is reflexive if every element of  $A$  is related to itself. For example, consider the relation Address defined as "lives at same address as". Address is a relation over a set of people. Clearly every person lives at the same address as him or herself, so Address is reflexive. So is the Less than or equal relation on the integers. Every integer is Less than or equal to itself. But the Less than relation is not reflexive: in fact no number is Less than itself. Both the directed graph and the matrix representations make it easy to tell if a relation is reflexive. In the graph representation, every node will have an edge looping back to itself. In the graph representation, there will be ones along the major diagonal:



1		
	1	
		1

A relation  $R \subseteq A \times A$  is **symmetric** if, whenever  $(a, b) \in R$ , so is  $(b, a)$ . In other words, if  $a$  is related to  $b$ , then  $b$  is related to  $a$ . The Address relation we described above is symmetric. If Joe lives with Ann, then Ann lives with Joe. The Less than or equal relation is not symmetric (since, for example,  $2 \leq 3$ , but it is not true that  $3 \leq 2$ ). The graph representation of a symmetric relation has the property that between any two nodes, either there is an arrow going in both directions or there is an arrow going in neither direction. So we get graphs with components that look like this:



If we choose the matrix representation, we will end up with a symmetric matrix (i.e., if you flip it on its major diagonal, you'll get the same matrix back again). In other words, if we have a matrix with 1's wherever there is a number in the following matrix, then there must also be 1's in all the squares marked with an \*:

	*	*		
1				3
2				
	*			

A relation  $R \subseteq A \times A$  is **antisymmetric** if, whenever  $(a, b) \in R$  and  $a \neq b$ , then  $(b, a) \notin R$ . The Mother-of relation we described above is antisymmetric: if Ann is the mother of Catherine, then one thing we know for sure is that Catherine is not also the mother of Ann. Our Address relation is clearly not antisymmetric, since it is symmetric. There are, however, relations that are neither symmetric nor antisymmetric. For example, the Likes relation on the set of people: If Joe likes Bob, then it is possible that Bob likes Joe, but it is also possible that he doesn't.

A relation  $R \subseteq A \times A$  is **transitive** if, whenever  $(a, b) \in R$  and  $(b, c) \in R$ ,  $(a, c) \in R$ . A simple example of a transitive relation is Less than. Address is another one: if Joe lives with Ann and Ann lives with Mark, then Joe lives with Mark. Mother-of is not transitive. But if we change it slightly to Ancestor-of, then we get a transitive relation. If Doreen is an ancestor of Ann and Ann is an ancestor of Catherine, then Doreen is an ancestor of Catherine.

The three properties of reflexivity, symmetry, and transitivity are almost logically independent of each other. We can find simple, possibly useful relationships with seven of the eight possible combinations of these properties:

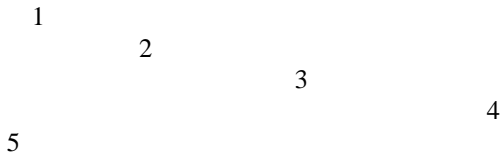
	Domain	Example
None of the properties	people	Mother-of
Just reflexive	people	Would-recognize-picture-of
Just symmetric	people	Has-ever-been-married-to
Just transitive	people	Ancestor-of
Reflexive and symmetric	people	Hangs-out-with (assuming we can say one hangs out with oneself)
Reflexive and transitive	numbers	Less than or equal to
Symmetric and transitive		
All three	numbers	Equality
	people	Address

To see why we can't find a good example of a relation that is symmetric and transitive but not reflexive, consider a simple relation  $R$  on  $\{1, 2, 3, 4\}$ . As soon as  $R$  contains a single element that relates two unequal objects (e.g.,  $(1, 2)$ ), it must, for symmetry, contain the matching element  $(2, 1)$ . So now we have  $R = \{(1, 2), (2, 1)\}$ . To make  $R$  transitive, we must add  $(1, 1)$ . But that also makes  $R$  reflexive.

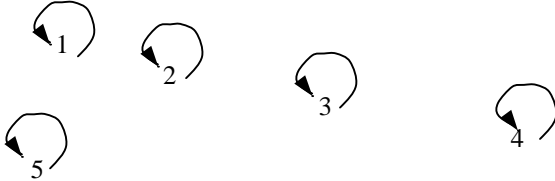
### 3.3 Equivalence Relations

Although all the combinations we just described are possible, one combination is of such great importance that we give it a special name. A relation is an **equivalence relation** if it is reflexive, symmetric and transitive. Equality (for numbers, strings, or whatever) is an equivalence relation (what a surprise, given the name). So is our Address (lives at same address) relation.

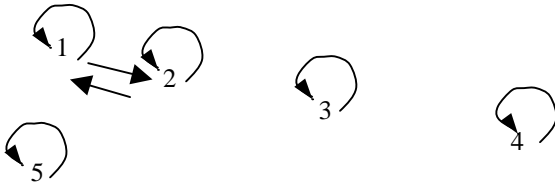
Equality is a very special sort of equivalence relation because it relates an object only to itself. It doesn't help us much to carve up a large set into useful subsets. But in fact, equivalence relations are an incredibly useful way to carve up a set. Why? Let's look at a set  $P$ , with five elements, which we can draw as a set of nodes as follows:



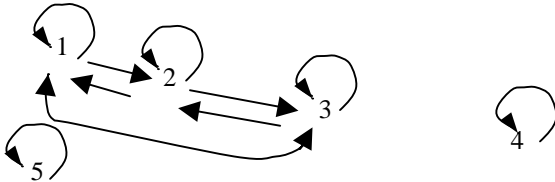
Now let's build an equivalence relation  $E$  on  $P$ . The first thing we have to do is to relate each node to itself, in order to make the relation reflexive. So we've now got:



Now let's add one additional element  $(1,2)$ . As soon as we do that, we must also add  $(2,1)$ , since  $E$  must be symmetric. So now we've got:



Suppose we now add  $(2,3)$ . We must also add  $(3,2)$  to maintain symmetry. In addition, because we have  $(1,2)$  and  $(2,3)$ , we must create  $(1,3)$  for transitivity. And then we need  $(3,1)$  to restore symmetry. That gives us



Notice what happened here. As soon as we related 3 to 2, we were also forced to relate 3 to 1. If we hadn't, we would no longer have had an equivalence relation. See what happens now if you add  $(3,4)$  to  $E$ .

What we've seen in this example is that an equivalence relation  $R$  on a set  $S$  carves  $S$  up into a set of clusters, which we'll call **equivalence classes**. This set of equivalence classes has the following key property:

For any  $s, t \in S$ , if  $s \in \text{Class}_i$  and  $(s, t) \in R$ , then  $t \in \text{Class}_i$ .

In other words, all elements of  $S$  that are related under  $R$  are in the same equivalence class. To describe equivalence classes, we'll use the notation  $[a]$  to mean the equivalence class to which  $a$  belongs. Or we may just write  $[\text{description}]$ , where description is some clear property shared by all the members of the class. Notice that in general there may be lots of different ways to describe the same equivalence class. In our example, for instance,  $[1]$ ,  $[2]$ , and  $[3]$  are different names for the same equivalence class, which includes the elements 1, 2, and 3. In this example, there are two other equivalence classes as well:  $[4]$  and  $[5]$ .

It is possible to prove that if  $R$  is an equivalence relation on a nonempty set  $A$  then the equivalence classes of  $R$  constitute a partition of  $A$ . Recall that  $\Pi$  is a partition of a set  $A$  if and only if (a) no element of  $\Pi$  is empty; (b) all members of  $\Pi$  are disjoint; and (c)  $\bigcup \Pi = A$ . In other words, if we want to take a set  $A$  and carve it up into a set of subsets, an equivalence relation is a good way to do it.

For example, our Address relation carves up a set of people into subsets of people who live together. Let's look at some more examples:

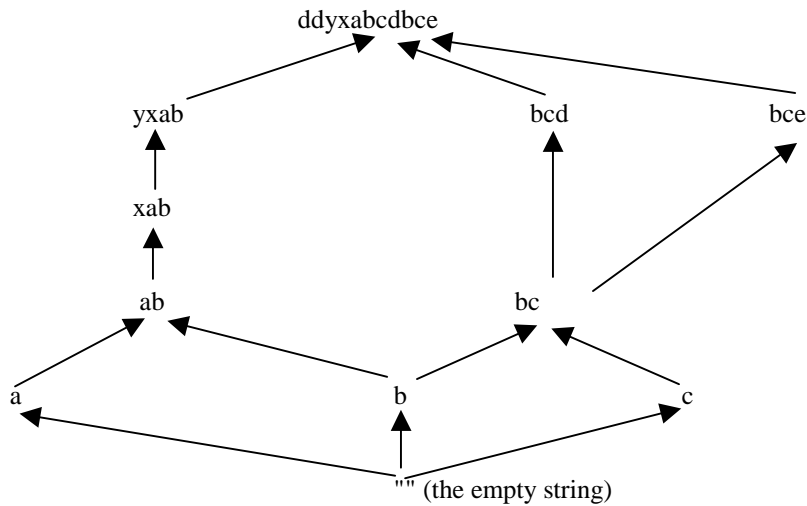
- Let  $A$  be the set of all strings of letters. Let  $\text{SameLength} \subseteq A \times A$  relate strings whose lengths are the same.  $\text{SameLength}$  is an equivalence relation that carves up the universe of all strings into a collection of subsets, one for each natural number (i.e., strings of length 0, strings of length 1, etc.).
- Let  $Z$  be the set of integers. Let  $\text{EqualMod3} \subseteq Z \times Z$  relate integers that have the same remainder when divided by 3.  $\text{EqualMod3}$  has three equivalence classes,  $[0]$ ,  $[1]$ , and  $[2]$ .  $[0]$  includes 0, 3, 6, etc.
- Let  $CP$  be the set of C programs, each of which accepts an input of variable length. We'll call the length of any specific input  $n$ . Let  $\text{SameComplexity} \subseteq CP \times CP$  relate two programs if their running-time complexity is the same. More specifically,  $(c_1, c_2) \in \text{SameComplexity}$  precisely in case:
 
$$\exists m_1, m_2, k [\forall n > k, \text{RunningTime}(c_1) \leq m_1 * \text{RunningTime}(c_2) \text{ AND } \text{RunningTime}(c_2) \leq m_2 * \text{RunningTime}(c_1)]$$

Not every relation that connects "similar" things is an equivalence relation. For example, consider  $\text{SimilarCost}(x, y)$ , which holds if the price of  $x$  is within \$1 of the price of  $y$ . Suppose  $A$  costs \$10,  $B$  costs \$10.50, and  $C$  costs \$11.25. Then  $\text{SimilarCost}(A, B)$  and  $\text{SimilarCost}(B, C)$ , but not  $\text{SimilarCost}(A, C)$ . So  $\text{SimilarCost}$  is not transitive, although it is reflexive and symmetric.

### 3.4 Orderings

Important as equivalence relations are, they're not the only special kind of relation worth mentioning. Let's consider two more.

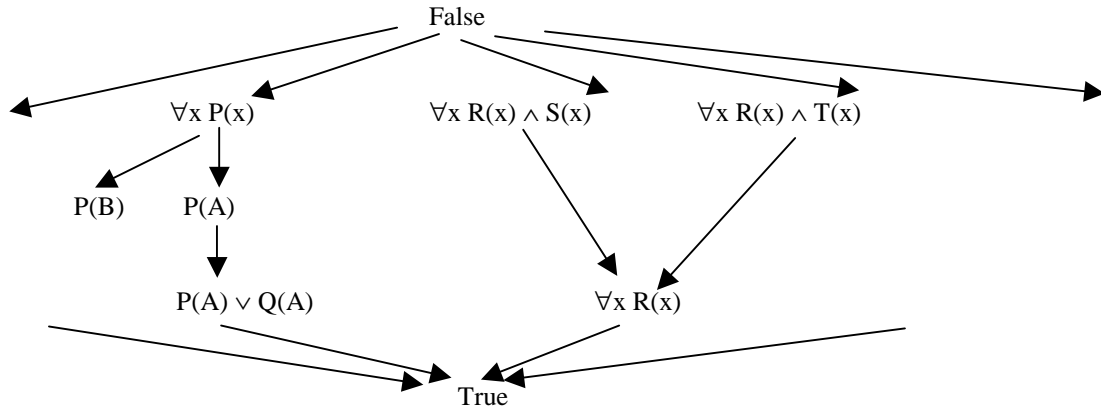
A **partial order** is a relation that is reflexive, antisymmetric, and transitive. If we write out any partial order as a graph, we'll see a structure like the following one for the relation  $\text{SubstringOf}$ . Notice that in order to make the graph relatively easy to read, we'll adopt the convention that we don't write in the links that are required by reflexivity and transitivity. But, of course, they are there in the relation itself:



Read an arrow from  $x$  to  $y$  as meaning that  $(x, y)$  is an element of the relation. So, in this example, "a" is a substring of "ab", which is a substring of "xab", and so forth. Note that in a partial order, it is often the case that there are some elements (such as "ab" and "bc") that are not related to each other at all (since neither is a substring of the other).

Sometimes a partial order on a domain  $D$  defines a minimal and/or a maximal element. In this example, there is a minimal element, the empty string, which is a substring of every string. There appears from this picture to be a maximal element, but that's just because we drew only a tiny piece of the graph. Every string is a substring of some longer string, so there is in fact no maximal element.

Let's consider another example based on the *subsumption* relation between pairs of logical expressions. A logical expression A subsumes another expression B iff (if and only if), whenever A is true B must be true regardless of the values assigned to the variables and functions of A and B. For example:  $\forall x P(x)$  subsumes  $P(A)$ , since, regardless of what the predicate P is and independently of any axioms we have about it, and regardless of what object A represents, if  $\forall x P(x)$  is true, then  $P(A)$  must be true. Why is this a useful notion? Suppose we're building a theorem proving or reasoning program. If we already know  $\forall x P(x)$ , and we are then told  $P(A)$ , we can throw away this new fact. It doesn't add to our knowledge (except perhaps to focus our attention on the object A) since it is subsumed by something we already knew. A small piece of the subsumption relation on logical expressions is shown in the following graph. Notice that now there is a maximal element, False, which subsumes everything (in other words, if we have the assertion False in our knowledge base, we have a contradiction even if we know nothing else). There is also a minimal element, True, which tells us nothing.



A **total order**  $R \subseteq A \times A$  is a partial order that has the additional property that  $\forall a, b \in A$ , either  $(a, b) \in R$  or  $(b, a) \in R$ . In other words, every pair of elements must be related to each other one way or another. The classic example of a total order is  $\leq$  (or  $\geq$ , if you prefer) on the integers. The  $\leq$  relation is reflexive since every integer is equal to itself. It's antisymmetric since if  $a \leq b$  and  $a \neq b$ , then for sure it is not also true that  $b \leq a$ . It's transitive: if  $a \leq b$  and  $b \leq c$ , then  $a \leq c$ . And, given any two integers a and b, either  $a \leq b$  or  $b \leq a$ . If we draw any total order as a graph, we'll get something that looks like this (again without the reflexive and transitive links shown):



This is only a tiny piece of the graph, of course. It continues infinitely in both directions. But notice that, unlike our earlier examples of partial orders, there is no splitting in this graph. For every pair of elements, one is above and one is below.

## 4 Important Properties of Binary Functions

Any relation that uniquely maps from all elements of its domain to elements of its range is a function. The two sets involved can be anything and the mapping can be arbitrary. However, most of the functions we actually care about behave in some sort of regular fashion. It is useful to articulate a set of properties that many of the functions that we'll study have. When these properties are true of a function, or a set of functions, they give us techniques for proving additional properties of the objects involved. In the following definitions, we'll consider an arbitrary binary function # defined over a set we'll call A with elements we'll call a, b, and c. As examples, we'll consider functions whose actual domains are sets, integers, strings, and boolean expressions.

A binary function # is **commutative** iff  $\forall a,b \ a \# \ b = b \# \ a$   
 Examples:  $a + b = b + a$  integer addition  
 $a \cap b = b \cap a$  set intersection  
 $a \text{ AND } b = b \text{ AND } a$  boolean and

A binary function # is **associative** iff  $\forall a,b,c \ (a \# \ b) \# \ c = a \# \ (b \# \ c)$   
 Examples:  $(a + b) + c = a + (b + c)$  integer addition  
 $(a \cap b) \cap c = a \cap (b \cap c)$  set intersection  
 $(a \text{ AND } b) \text{ AND } c = a \text{ AND } (b \text{ AND } c)$  boolean and  
 $(a \parallel b) \parallel c = a \parallel (b \parallel c)$  string concatenation

A binary function # is **idempotent** iff  $\forall a \ a \# \ a = a.$   
 Examples:  $\min(a, a) = a$  integer min  
 $a \cap a = a$  set intersection  
 $a \text{ AND } a = a$  boolean and

The **distributivity** property relates two binary functions: A function # distributes over another function ! iff  $\forall a,b,c \ a \# \ (b \ ! \ c) = (a \# \ b) \ ! \ (a \# \ c)$  and  $(b \ ! \ c) \# \ a = (b \# \ a) \ ! \ (c \# \ a)$

Examples:  $a * (b + c) = (a * b) + (a * c)$  integer multiplication over addition  
 $a \cup (b \cap c) = (a \cup b) \cap (a \cup c)$  set union over intersection  
 $a \text{ AND } (b \text{ OR } c) = (a \text{ AND } b) \text{ OR } (a \text{ AND } c)$  boolean AND over OR

The **absorption laws** also relate two binary functions to each other: A function # absorbs another function ! iff  $\forall a,b \ a \# \ (a \ ! \ b) = a$

Examples:  $a \cap (a \cup b) = a$  set intersection absorbs union  
 $a \text{ OR } (a \text{ AND } b) = a$  boolean OR absorbs AND

It is often the case that when a function is defined over some set A, there are special elements of A that have particular properties with respect to that function. In particular, it is worth defining what it means to be an identity and to be a zero:

An element a is an **identity** for the function # iff  $\forall x \in A, \ x \# \ a = x$  and  $a \# \ x = x$

Examples:  $b * 1 = b$  1 is an identity for integer multiplication  
 $b + 0 = b$  0 is an identity for integer addition  
 $b \cup \emptyset = b$   $\emptyset$  is an identity for set union  
 $b \text{ OR } \text{False} = b$  False is an identity for boolean OR  
 $b \parallel "" = b$  "" is an identity for string concatenation

Sometimes it is useful to differentiate between a right identity (one that satisfies the first requirement above) and a left identity (one that satisfies the second requirement above). But for all the functions we'll be concerned with, if there is a left identity, it is also a right identity and vice versa, so we will talk simply about an identity.

An element a is a **zero** for the function # iff  $\forall x \in A, \ x \# \ a = a$  and  $a \# \ x = a$

Examples:  $b * 0 = 0$  0 is a zero for integer multiplication  
 $b \cap \emptyset = \emptyset$   $\emptyset$  is a zero for set intersection  
 $b \text{ AND } \text{FALSE} = \text{FALSE}$  FALSE is a zero for boolean AND

Just as with identities, it is sometimes useful to distinguish between left and right zeros, but we won't need to.

Although we're focusing here on binary functions, there's one important property that unary functions may have that is worth mentioning here:

A unary function  $%$  is a **self inverse** iff  $\forall x \ %(%x) = x$ . In other words, if we compose the function with itself (apply it twice), we get back the original argument. Note that this is not the same as saying that the function is its own inverse. In most of the cases we'll consider (including the examples given here), it is not. A single application of the function produces a new value, but if we apply the function a second time, we get back to where we started.

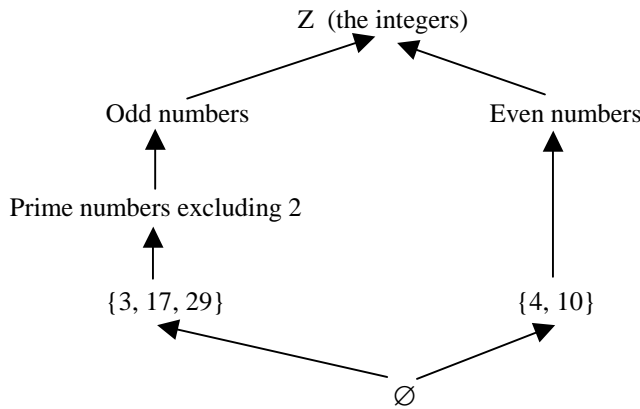
<p>Examples:</p> $-(-a) = a$ $\frac{1}{1/a}$ $\bar{\bar{a}} = a$ $\neg(\neg a) = a$ $(a^R)^R = a$	<p>Multiplying by -1 is a self inverse for integers          Dividing into 1 is a self inverse for integers          Complement is a self inverse for sets          Negation is a self inverse for booleans          Reversal is a self inverse for strings</p>
---	---

## 5 Relations and Functions on Sets

In the last two sections, we explored various useful properties of relations and functions. With those tools in hand, let's revisit the basic relations and functions on sets.

### 5.1 Relations

We have defined two relations on sets: subset and proper subset. What can we say about them? Subset is a partial order, since it is reflexive (every set is a subset of itself), transitive (if  $A \subseteq B$  and  $B \subseteq C$ , then  $A \subseteq C$ ) and antisymmetric (if  $A \subseteq B$  and  $A \neq B$ , then it must not be true that  $B \subseteq A$ ). For example, we see that the subset relation imposes the following partial order if you read each arrow as "is a subset of":



What about proper subset? It is not a partial order since it is not reflexive.

### 5.2 Functions

All of the functional properties we defined above apply in one way or another to the functions we have defined on sets. Further, as we saw above, there some set functions have a zero or an identity. We'll summarize here (without proof) the most useful properties that hold for the functions we have defined on sets:

**Commutativity**

$$A \cup B = B \cup A$$

$$A \cap B = B \cap A$$

**Associativity**

$$(A \cup B) \cup C = A \cup (B \cup C)$$

$$(A \cap B) \cap C = A \cap (B \cap C)$$

<b>Idempotency</b>	$A \cup A = A$ $A \cap A = A$
<b>Distributivity</b>	$(A \cup B) \cap C = (A \cap C) \cup (B \cap C)$ $(A \cap B) \cup C = (A \cup C) \cap (B \cup C)$
<b>Absorption</b>	$(A \cup B) \cap A = A$ $(A \cap B) \cup A = A$
<b>Identity</b>	$A \cup \emptyset = A$
<b>Zero</b>	$A \cap \emptyset = \emptyset$
<b>Self Inverse</b>	$\overline{\overline{A}} = A$

In addition, we will want to make use of the following theorems that can be proven to apply specifically to sets and their operations (as well as to boolean expressions):

**De Morgan's laws** 
$$\overline{A \cup B} = \overline{A} \cap \overline{B}$$
$$\overline{A \cap B} = \overline{A} \cup \overline{B}$$

## 6 Proving Properties of Sets

A great deal of what we do when we build a theory about some domain is to prove that various sets of objects in that domain are equal. For example, in our study of automata theory, we are going to want to prove assertions such as these:

- The set of strings defined by some regular expression E is identical to the set of strings defined by some second regular expression E'.
- The set of strings that will be accepted by some given finite state automaton M is the same as the set of strings that will be accepted by some new finite state automaton M' that is smaller than M.
- The set of languages that can be defined using regular expressions is the same as the set of languages that can be accepted by a finite state automaton.
- The set of problems that can be solved by a Turing Machine with a single tape is the same as the set of problems that can be solved by a Turing Machine with any finite number of tapes.

So we become very interested in the question, "How does one prove that two sets are identical"? There are lots of ways and many of them require special techniques that apply in specific domains. But it's worth mentioning two very general approaches here.

### 6.1 Using Set Identities and the Definitions of the Functions on Sets

Sometimes we want to compare apples to apples. We may, for example, want to prove that two sets of strings are identical, even though they may have been derived differently. In this case, one approach is to use the set identity theorems that we enumerated in the last section. Suppose, for example, that we want to prove that

$$A \cup (B \cap (A \cap C)) = A$$

We can prove this as follows:

$A \cup (B \cap (A \cap C)) = (A \cup B) \cap (A \cup (A \cap C))$	Distributivity
$= (A \cup B) \cap ((A \cap C) \cup A)$	Commutativity
$= (A \cup B) \cap A$	Absorption
$= A$	Absorption

Sometimes, even when we're comparing apples to apples, the theorems we've listed aren't enough. In these cases, we need to use the definitions of the operators. Suppose, for example, that we want to prove that

$$A - B = A \cap \overline{B}$$

We can prove this as follows (where U stands for the Universe with respect to which we take complement):



$$\begin{aligned}
A - B &= \{x : x \in A \text{ and } x \notin B\} \\
&= \{x : x \in A \text{ and } (x \in U \text{ and } x \notin B)\} \\
&= \{x : x \in A \text{ and } x \in U - B\} \\
&= \{x : x \in A \text{ and } x \in \bar{B}\} \\
&= A \cap \bar{B}
\end{aligned}$$

## 6.2 Showing Two Sets are the Same by Showing that Each is a Subset of the Other

Sometimes, though, our problem is more complex. We may need to compare apples to oranges, by which I mean that we are comparing sets that aren't even defined in the same terms. For example, we will want to be able to prove that A: the set of languages that can be defined using regular expressions is the same as B: the set of languages that can be accepted by a finite state automaton. This seems very hard: Regular expressions look like

$$a^*(b \cup ba)^*$$

Finite state machines are a collection of states and rules for moving from one state to another. How can we possibly prove that these A and B are the same set? The answer is that we can show that the two sets are equal by showing that each is a subset of the other. For example, in the case of the regular expressions and the finite state machines, we will show first that, given a regular expression, we can construct a finite state machine that accepts exactly the strings that the regular expression describes. That gives us  $A \subseteq B$ . But there might still be some finite state machines that don't correspond to any regular expressions. So we then show that, given a finite state machine, we can construct a regular expression that defines exactly the same strings that the machine accepts. That gives us  $B \subseteq A$ . The final step is to exploit the fact that

$$A \subseteq B \text{ and } B \subseteq A \Rightarrow A = B$$

## 7 Cardinality of Sets

It seems natural to ask, given some set A, "What is the size of A?" or "How many elements does A contain?" In fact, we've been doing that informally. We'll now introduce formal techniques for discussing exactly what we mean by the size of a set. We'll use the term *cardinality* to describe the way we answer such questions. So we'll reply that the cardinality of A is X, for some appropriate value of X. For simple cases, determining the value of X is straightforward. In other cases, it can get quite complicated. For our purposes, however, we can get by with three different kinds of answers: a natural number (if A is finite), "countably infinite" (if A has the same number of elements as there are integers), and "uncountably infinite" (if A has more elements than there are integers).

We write the cardinality of a set A as  $|A|$ .

A set A is *finite* and has cardinality  $n \in \mathbb{N}$  (the natural numbers) if either  $A = \emptyset$  or there is a bijection from A to  $\{1, 2, \dots, n\}$ , for some value of n. In other words, a set is finite if either it is empty or there exists a one-to-one and onto mapping from it to a subset of the positive integers. Or, alternatively, a set is finite if we can count its elements and finish. The cardinality of a finite set is simply a natural number whose value is the number of elements in the set.

A set is *infinite* if it is not finite. The question now is, "Are all infinite sets the same size?" The answer is no. And we don't have to venture far to find examples of infinite sets that are not the same size. So we need some way to describe the cardinality of infinite sets. To do this, we need to define a set of numbers we'll call the cardinal numbers. We'll use these numbers as our measure of the size of sets. Initially, we'll define all the natural numbers to be cardinal numbers. That lets us describe the cardinality of finite sets. Now we need to add new cardinal numbers to describe the cardinality of infinite sets.

Let's start with a simple infinite set  $\mathbb{N}$ , the natural numbers. We need a new cardinal number to describe the (infinite) number of natural numbers that there are. Following Cantor, we'll call this number  $\aleph_0$ . (Read this as "aleph null". Aleph is the first symbol of the Hebrew alphabet.)

Next, we'll say that any other set that contains the same number of members as  $\mathbb{N}$  does also has cardinality  $\aleph_0$ . We'll also call a set with cardinality  $\aleph_0$  *countably infinite*. And one more definition: A set is *countable* if it is either finite or countably infinite.

To show that a set has cardinality  $\aleph_0$ , we need to show that there is a bijection between it and  $\mathbb{N}$ . The existence of such a bijection proves that the two sets have the same number of elements. For example, the set  $E$  of even natural numbers has cardinality  $\aleph_0$ . To prove this, we offer the bijection:

$$\begin{aligned} \text{Even} : E &\rightarrow \mathbb{N} \\ \text{Even}(x) &= x/2 \end{aligned}$$

So we have the following mapping from  $E$  to  $\mathbb{N}$ :

E	N
0	0
2	1
4	2
6	3
...	...

This one was easy. The bijection was obvious. Sometimes it's less so. In harder cases, a good way to think about the problem of finding a bijection from some set  $A$  to  $\mathbb{N}$  is that we need to find an enumeration of  $A$ . An *enumeration* of  $A$  is simply a list of the elements of  $A$  in some order. Of course, if  $A$  is infinite, the list will be infinite, but as long as we can guarantee that every element of  $A$  will show up eventually, we have an enumeration. But what is an enumeration? It is in fact a bijection from  $A$  to the positive integers, since there is a first element, a second one, a third one, and so forth. Of course, what we need is a bijection to  $\mathbb{N}$ , so we just subtract one. Thus if we can devise a technique for enumerating the elements of  $A$ , then our bijection to  $\mathbb{N}$  is simply

$$\begin{aligned} \text{Enum} : A &\rightarrow \mathbb{N} \\ \text{Enum}(x) &= x\text{'s position in the enumeration} - 1 \end{aligned}$$

Let's consider an example of this technique:

**Theorem:** The union of a countably infinite number of countably infinite sets is countably infinite.

To prove this theorem, we need a way to enumerate all the elements of the union. The simplest thing to do would be to start by dumping in all the elements of the first set, then all the elements of the second, etc. But, since the first set is infinite, we'll never get around to considering any of the elements of the other sets. So we need another technique. If we had a finite number of sets to consider, we could take the first element from each, then the second element from each, and so forth. But we also have an infinite number of sets, so if we try that approach, we'll never get to the second element of any of the sets. So we follow the arrows as shown below. The numbers in the squares indicate the order in which we select elements for the enumeration. This process goes on forever, but it is systematic and it guarantees that, if we wait long enough, any element of any of the sets will eventually be enumerated.

	Set 1	Set 2	Set 3	Set 4	...
Element 1	1 ↓	3 →	4 →	→	→
Element 2	2 ↓	5 →	→	→	→
Element 3	6 ↓	8 →	→	→	→
...	7 ↓	→	→	→	→

It turns out that a lot of sets have cardinality  $\aleph_0$ . Some of them, like the even natural numbers, appear at first to contain fewer elements. Some of them, like the union of a countable number of countable sets, appear at first to be bigger. But in both cases there is a bijection between the elements of the set and the natural numbers, so the cardinality is  $\aleph_0$ .

However, this isn't true for every set. There are sets with more than  $\aleph_0$  elements. There are more than  $\aleph_0$  real numbers, for example. As another case, consider an arbitrary set  $S$  with cardinality  $\aleph_0$ . Now consider the power set of  $S$  (the set of all subsets of  $S$ ). This set has cardinality greater than  $\aleph_0$ . To prove this, we need to show that there exists no bijection

between the power set of  $S$  and the integers. To do this, we will use a technique called *diagonalization*. Diagonalization is a kind of proof by contradiction. It works as follows:

Let's start with the original countably infinite set  $S$ . We can enumerate the elements of  $S$  (since it's countable), so there's a first one, a second one, etc. Now we can represent each subset  $SS$  of  $S$  as a binary vector that contains one element for each element of the original set  $S$ . If  $SS$  contains element 1 of  $S$ , then the first element of its vector will be 1, otherwise 0. Similarly for all the other elements of  $S$ . Of course, since  $S$  is countably infinite, the length of each vector will also be countably infinite. Thus we might represent a particular subset  $SS$  of  $S$  as the vector:

Elem 1 of S	Elem 2 of S	Elem 3 of S	Elem 4 of S	Elem 5 of S	Elem 6 of S	.....
1	0	0	1	1	0	.....

Next, we observe that if the power set  $P$  of  $S$  were countably infinite, then there would be an enumeration of it that put its elements in one to one correspondence with the natural numbers. Suppose that enumeration were the following (where each row represents one element of  $P$  as described above. Ignore for the moment the numbers enclosed in parentheses.):

	Elem 1 of S	Elem 2 of S	Elem 3 of S	Elem 4 of S	Elem 5 of S	Elem 6 of S	.....
Elem 1 of P	1 (1)	0	0	0	0	0	.....
Elem 2 of P	0	1 (2)	0	0	0	0	.....
Elem 3 of P	1	1	0 (3)	0	0	0	.....
Elem 4 of P	0	0	1	0 (4)	0	0	.....
Elem 5 of P	1	0	1	0	0 (5)	0	.....
Elem 6 of P	1	1	1	0	0	0 (6)	.....



If this really is an enumeration of  $P$ , then it must contain all elements of  $P$ . But it doesn't. To prove that it doesn't, we will construct an element  $L \in P$  that is not on the list. To do this, consider the numbers in parentheses in the matrix above. Using them, we can construct  $L$ :

$\neg(1)$	$\neg(2)$	$\neg(3)$	$\neg(4)$	$\neg(5)$	$\neg(6)$	.....
-----------	-----------	-----------	-----------	-----------	-----------	-------

What we mean by  $\neg(1)$  is that if (1) is a 1 then 0; if (1) is a 0, then 1. So we've constructed the representation for an element of  $P$ . It must be an element of  $P$  since it describes a possible subset of  $S$ . But we've built it so that it differs from the first element in the list above by whether or not it includes element 1 of  $S$ . It differs from the second element in the list above by whether or not it includes element 2 of  $S$ . And so forth. In the end, it must differ from every element in the list above in at least one place. Yet it is clearly an element of  $P$ . Thus we have a contradiction. The list above was not an enumeration of  $P$ . But since we made no assumptions about it, no enumeration of  $P$  can exist. In particular, if we try to fix the problem by simply adding our new element to the list, we can just turn around and do the same thing again and create yet another element that's not on the list. Thus there are more than  $\aleph_0$  elements in  $P$ . We'll say that sets with more than  $\aleph_0$  elements are *uncountably infinite*.

The real numbers are uncountably infinite. The proof that they are is very similar to the one we just did for the power set except that it's a bit tricky because, when we write out each number as an infinite sequence of digits (like we wrote out each set above as an infinite sequence of 0's and 1's), we have to consider the fact that several distinct sequences may represent the same number.

Not all uncountably infinite sets have the same cardinality. There is an infinite number of cardinal numbers. But we won't need any more. All the uncountably infinite sets we'll deal with (and probably all the ones you can even think of unless you keep taking power sets) have the same cardinality as the reals and the power set of a countably infinite set.

Thus to describe the cardinality of all the sets we'll consider, we will use one of the following:

- The natural numbers, which we'll use to count the number of elements of finite sets,

- $\aleph_0$ , which is the cardinality of all countably infinite sets, and
- uncountable, which is the cardinality of any set with more than  $\aleph_0$  members.

## 8 Closures

Imagine some set  $A$  and some property  $P$ . If we care about making sure that  $A$  has property  $P$ , we are likely to do the following:

1. Examine  $A$  for  $P$ . If it has property  $P$ , we're happy and we quit.
2. If it doesn't, then add to  $A$  the smallest number of additional elements required to satisfy  $P$ .

Let's consider some examples:

- Let  $A$  be a set of friends you're planning to invite to a party. Let  $P$  be "A should include everyone who is likely to find out about the party" (since we don't want to offend anyone). Let's assume that if you invite Bill and Bill has a friend Bob, then Bill may tell Bob about the party. This means that if you want  $A$  to satisfy  $P$ , then you have to invite not only your friends, but your friends' friends, and their friends, and so forth. If you move in a fairly closed circle, you may be able to satisfy  $P$  by adding a few people to the guest list. On the other hand, it's possible that you'd have to invite the whole city before  $P$  would be satisfied. It depends on the connectivity of the FriendsOf relation in your social setting. The problem is that whenever you add a new person to  $A$ , you have to turn around and look at that person's friends and consider whether there are any of them who are not already in  $A$ . If there are, they must be added, and so forth. There's one positive feature of this problem, however. Notice that there is a unique set that does satisfy  $P$ , given the initial set  $A$ . There aren't any choices to be made.
- Let  $A$  be a set of 6 people. Let  $P$  be "A can enter a baseball tournament". This problem is different from the last in two important ways. First, there is a clear limit to how many elements we have to add to  $A$  in order to satisfy  $P$ . We need 9 people and when we've got them we can stop. But notice that there is not a unique way to satisfy  $P$  (assuming that we know more than 9 people). Any way of adding 3 people to the set will work.
- Let  $A$  be the Address relation (which we defined earlier as "lives at same address as"). Since relations are sets, we should be able to treat Address just as we've treated the sets of people in our last two examples. We know that Address is an equivalence relation. So we'll let  $P$  be the property of being an equivalence relation (i.e., reflexive, symmetric, and transitive). But suppose we are only able to collect facts about living arrangements in a piecemeal fashion. For example, we may learn that Address contains  $\{(Dave, Mary), (Sue, Pete), (John, Bill)\}$ . Immediately we know, because Address must be reflexive, that it must also contain  $\{(Dave, Dave), (Mary, Mary), (Sue, Sue), (Pete, Pete), (John, John), (Bill, Bill)\}$ . And, since Address must also be symmetric it must contain  $\{(Mary, Dave), (Pete, Sue), (Bill, John)\}$ . Now suppose that we discover that Mary lives with Sue. We add  $\{(Mary, Sue)\}$ . To make Address symmetric again, we must add  $\{(Sue, Mary)\}$ . But now we also have to make it transitive by adding  $\{(Dave, Sue), (Sue, Dave)\}$ .
- Let  $A$  be the set of natural numbers. Let  $P$  be "the sum of any two elements of  $A$  is also in  $A$ ." Now we've got a property that is already satisfied. The sum of any two natural numbers is a natural number. This time, we don't have to add anything to  $A$  to establish  $P$ .
- Let  $A$  be the set of natural numbers. Let  $P$  be "the quotient of any two elements of  $A$  is also in  $A$ ." This time we have a problem.  $3/5$  is not a natural number. We can add elements to  $A$  to satisfy  $P$ . If we do, we end up with exactly the rational numbers.

In all of these cases, we're going to want to say that  $A$  is *closed* with respect to  $P$  if it possesses  $P$ . And, if we have to add elements to  $A$  in order to satisfy  $P$ , we'll call a smallest such expanded  $A$  that does satisfy  $P$  a *closure* of  $A$  with respect to  $P$ . What we need to do next is to define both of these terms more precisely.

### 8.1 Defining Closure

The first set of definitions of closure that we'll present is very general, although it does require that we can describe property  $P$  as an  $n$ -ary relation (for some value of  $n$ ). We can use it to describe what we did in all but one of the examples above, although in a few cases it will be quite cumbersome to do so.

Let  $n$  be an integer greater than or equal to 1. Let  $R$  be an  $n$ -ary relation on a set  $D$ . Thus elements of  $R$  are of the form  $(d_1, d_2, \dots, d_n)$ . We say that a subset  $S$  of  $D$  is *closed under*  $R$  if, whenever:

1.  $d_1, d_2, \dots, d_{n-1} \in S$ , (all of the first  $n-1$  elements are already in the set  $S$ ) and
  2.  $(d_1, d_2, \dots, d_{n-1}, d_n) \in R$  (the last element is related to the  $n-1$  other elements via  $R$ )
- it is also true that  $d_n \in S$ .

A set  $S'$  is a **closure** of  $S$  with respect to  $R$  (defined on  $D$ ) iff:

1.  $S \subseteq S'$ ,
2.  $S'$  is closed under  $R$ , and
3.  $\forall T (T \subseteq D \text{ and } T \text{ is closed under } R) \Rightarrow |S'| \leq |T|$ .

In other words,  $S'$  is a closure of  $S$  with respect to  $R$  if it is an extension of  $S$  that is closed under  $R$  and if there is no smaller set that also meets both of those requirements. Note that we can't say that  $S'$  must be the smallest set that will do the job, since we do not yet have any guarantee that there is a unique such smaller set (recall the softball example above).

These definitions of closure are a very natural way to describe our first example above. Drawing from a set  $A$  of people, you start with  $S$  equal to your friends. Then, to compute your invitee list, you simply take the closure of  $S$  with respect to the relation `FriendOf`, which will force you to add to  $A$  your friends' friends, their friends, and so forth.

Now consider our second example, the case of the baseball team. Here there is no relation  $R$  that specifies, if one or more people are already on the team, then some specific other person must also be on. The property we care about is a property of the team (set) as a whole and not a property of patterns of individuals (elements). Thus this example, although similar, is not formally an instance of closure as we have just defined it. This turns out to be significant and leads us to the following definition:

Any property that asserts that a set  $S$  is closed under some relation  $R$  is a **closure property** of  $S$ . It is possible to prove that if  $P$  is a closure property, as just defined, on a set  $A$  and  $S$  is a subset of  $A$ , then the closure of  $S$  with respect to  $R$  exists and is unique. In other words, there exists a unique minimal set  $S'$  that contains  $S$  and is closed under  $R$ . Of all of our examples above, the baseball example is the only one that cannot be described in the terms of our definition of a closure property. The theorem that we have just stated (without proof) guarantees, therefore, that it will be the only one that does not have a unique minimal solution.

The definitions that we have just provided also work to describe our third example, in which we want to compute the closure of a relation (since, after all, a relation is a set). All we have to do is to come up with relations that describe the properties of being reflexive, symmetric, and transitive. To help us see what those relations need to be, let's recall our definitions of symmetry, reflexivity, and transitivity:

- A relation  $R \subseteq A \times A$  is **reflexive** if, for each  $a \in A$ ,  $(a, a) \in R$ .
- A relation  $R \subseteq A \times A$  is **symmetric** if, whenever  $(a, b) \in R$ , so is  $(b, a)$ .
- A relation  $R \subseteq A \times A$  is **transitive** if, whenever  $(a, b) \in R$  and  $(b, c) \in R$ ,  $(a, c) \in R$ .

Looking at these definitions, we can come up with three relations, Reflexivity, Symmetry, and Transitivity. All three are relations on relations, and they will enable us to define these three properties using the closure definitions we've given so far. All three definitions assume a base set  $A$  on which the relation we are interested is defined:

- $\forall a \in A, ((a, a) \in \text{Reflexivity})$ . Notice the double parentheses here. Reflexivity is a unary relation, where each element is itself an ordered pair. It doesn't really "relate" two elements. It is simply a list of ordered pairs. To see how it works to define reflexive closure, imagine a set  $A = \{x, y\}$ . Now suppose we start with a relation  $R$  on  $A = \{(x, y)\}$ . Clearly  $R$  isn't reflexive. And the Reflexivity relation tells us that it isn't because the reflexivity relation on  $A$  contains  $\{((x, x)), ((y, y))\}$ . This is a unary relation. So  $n$ , in the definition of closure, is 1. Consider the first element  $((x, x))$ . We consider all the components before the  $n$ th (i.e., first) and see if they're in  $A$ . This means we consider the first zero components. Trivially, all zero of them are in  $A$ . So the  $n$ th (the first) must also be. This means that  $(x, x)$  must be in  $R$ . But it isn't. So to compute the closure of  $R$  under Reflexivity, we add it. Similarly for  $(y, y)$ .
- $\forall a, b \in A, a \neq b \Rightarrow [((a, b), (b, a)) \in \text{Symmetry}]$ . This one is a lot easier. Again, suppose we start with a set  $A = \{x, y\}$  and a relation  $R$  on  $A = \{(x, y)\}$ . Clearly  $R$  isn't symmetric. And Symmetry tells us that. Symmetry on  $A = \{((x, y), (y, x)), ((y, x), (x, y))\}$ . But look at the first element of Symmetry. It tells us that for  $R$  to be closed, whenever  $(x, y)$  is in  $R$ ,  $(y, x)$  must also be. But it isn't. To compute the closure of  $R$  under Symmetry, we must add it.

- $\forall a, b, c \in A, [a \neq b \wedge b \neq c] \Rightarrow [(a, b), (b, c), (a, c)] \in \text{Transitivity}$ . Now we will exploit a ternary relation. Whenever the first two elements of it are present in some relation R, then the third must also be if R is transitive. This time, let's start with a set  $A = \{x, y, z\}$  and a relation  $R$  on  $A = \{(x, y), (y, z)\}$ . Clearly R is not transitive. The Transitivity relation on A is  $\{((x, y), (y, z), (x, z)), ((x, z), (z, y), (x, y)), ((y, x), (x, z), (y, z)), ((y, z), (z, x), (y, x)), ((z, x), (x, y), (z, y)), ((z, y), (y, x), (z, x))\}$ . Look at the first element of it. Both of the first two components of it are in R. But the third isn't. To make R transitive, we must add it.

These definitions also work to enable us to describe the closure of the integers under division as the rationals. Following the definition, A is the set of rationals. S (a subset of A) is the integers and R is QuotientClosure, defined as:

- $\forall a, b, c \in A, [a/b = c] \Rightarrow [(a, b, c)] \in \text{QuotientClosure}$ .

So we've got a quite general definition of closure. And it makes it possible to prove the existence of a unique closure for any set and any relation R. The only constraint is that this definition works only if we can define the property we care about as an n-ary relation for some finite n. There are cases of closure where this isn't possible, as we saw above, but we won't need to worry about them in this class.

So we don't really need any new definitions. We've offered a general definition of closure of a set (any set) under some relation (which is the way we use to define a property). But most of the cases of closure that we'll care about involve the special case of the closure of a binary relation given some property that may or may not be naturally describable as a relation. For example, one could argue that the relations we just described to define the properties of being reflexive, symmetric, and transitive are far from natural. Thus it will be useful to offer the following alternative definitions. Don't get confused though by the presence of two definitions. Except when we cannot specify our property P as a relation (and we won't need to deal with any such cases), these new definitions are simply special cases of the one we already have.

We say that a binary relation B on a set T is **closed under** property P if B possesses P. For example, LessThanOrEqualTo is closed under transitivity since it is transitive. Simple enough. Next:

Let B be a binary relation on a set T. A relation B' is a **closure** of B with respect to some property P iff:

1.  $B \subseteq B'$ ,
2. B' is closed under P, and
3. There is no smaller relation B'' that contains B and is closed under P.

So, for example, the transitive closure of  $B = \{(1, 2), (2, 3)\}$  is the smallest new relation B' that contains B but is transitive. So  $B' = \{(1, 2), (2, 3), (1, 3)\}$ .

You'll generally find it easier to use these definitions than our earlier ones. But keep in mind that, with the earlier definitions it is possible to prove the existence of a unique closure. Since we went through the process of defining reflexivity, symmetry, and transitivity using those definitions, we know that there always exists a unique reflexive, symmetric, and transitive closure for any binary relation. We can exploit that fact that the same time that we use the simpler definitions to help us find algorithms for computing those closures.

## 8.2 Computing Closures

Suppose we are given a set and a property and we want to compute the closure of the set with respect to the property. Let's consider two examples:

- Compute the symmetric closure of a binary relation B on a set A. This is trivial. Simply make sure that, for all elements x of A,  $(x, x) \in B$ .
- Compute the transitive closure of a binary relation B on a set A. This is harder. We can't just add a fixed number of elements to B and then quit. Every time we add a new element, such as  $(x, y)$ , we have to look to see whether there's some element  $(y, z)$  so now we also have to add  $(x, z)$ . And, similarly, we must check for any element  $(w, x)$  that would force us to add  $(w, y)$ . If A is infinite, there is no guarantee that this process will ever terminate. Our theorem that guaranteed that a unique closure exists did not guarantee that it would contain a finite number of elements and thus be computable in a finite amount of time.

We can, however, guarantee that the transitive closure of any binary relation on a *finite* set is computable. How? A very simple approach is the following algorithm for computing the transitive closure of a binary relation B with N elements on a set A:

```

Set Trans = B;                               /* Initially Trans is just the original relation.
/* We need to find all cases where (x, y) and (y, z) are in Trans. Then we must insert (x, z) into Trans if
/* it isn't already there.
Boolean AddedSomething = True;               /* We'll keep going until we make one whole pass through
                                              without adding any new elements to Trans.

while AddedSomething = True do
  AddedSomething = False;
  Xcounter := 0;
  Foreach element of Trans do
    xcounter := xcounter + 1;
    x = Trans[xcounter][1]                   /* Pull out the first element of the current element of Trans
    y = Trans[xcounter][2]                   /* Pull out the second element of the i'th element of Trans
                                              /* So if the first element of Trans is (p, q), then
                                              /* x = p and y = q the first time through.

    zcounter := 0;
    foreach element of Trans do
      zcounter := zcounter + 1;
      if Trans[zcounter][1] = y then do      /* We've found another element (y, ?) and we may need to
        z = Trans[zcounter][2];             /* add (x, ?) to Trans.
        if (x, z) ∉ Trans then do          /* we have to add it
          Insert(Trans, (x, z))
          AddedSomething = True;
    end;   end;   end;   end;   end;

```

This algorithm works. Try it on some simple examples. But it's very inefficient. There are much more efficient algorithms. In particular, if we represent a relation as an incidence matrix, we can do a lot better. Using Warshall's algorithm, for example, we can find the transitive closure of a relation of n elements using  $2n^3$  bit operations. For a description of that algorithm, see, for example, Kenneth Rosen, *Discrete Mathematics and its Applications*, McGraw-Hill.

## 9 Proof by Mathematical Induction

In the last section but one, as a sideline to our main discussion of cardinality, we presented diagonalization as a proof technique. In this section, we'll present one other very useful proof technique.

**Mathematical induction** is a technique for proving assertions about the set of positive integers. A proof by induction of assertion A about some set of positive integers greater than or equal to some specific value has two steps:

1. Prove that A holds for the smallest value we're concerned with. We'll call this value v. Generally  $v = 0$  or 1, but sometimes A may hold only once we get past some initial unusual cases.
2. Prove that  $\forall n \geq v, A(n) \Rightarrow A(n+1)$

We'll call  $A(n)$  the **induction hypothesis**. Since we're trying to prove that  $A(n) \Rightarrow A(n+1)$ , we can assume the induction hypothesis as an axiom in our proof.

Let's do a simple example and use induction to prove that the sum of the first n odd positive integers is  $n^2$ . Notice that this appears to be true:

$$\begin{aligned}
 (n = 1) \quad 1 &= 1 = 1^2 \\
 (n = 2) \quad 1 + 3 &= 4 = 2^2 \\
 (n = 3) \quad 1 + 3 + 5 &= 9 = 3^2 \\
 (n = 4) \quad 1 + 3 + 5 + 7 &= 16 = 4^2, \text{ and so forth.}
 \end{aligned}$$

To prove it, we need to follow the two steps that we just described:

1. Let  $v = 1$ :  $1 = 1^2$
2. Prove that,  $\forall n \geq 0$ ,

$$\left(\sum_{i=1}^n \text{Odd}_i = n^2\right) \Rightarrow \left(\sum_{i=1}^{n+1} \text{Odd}_i = (n+1)^2\right)$$

To do this, we observe that the sum of the first  $n+1$  odd integers is the sum of the first  $n$  of them plus the  $n+1$ 'st, i.e.,

$$\begin{aligned} \sum_{i=1}^{n+1} \text{Odd}_i &= \sum_{i=1}^n \text{Odd}_i + \text{Odd}_{n+1} \\ &= n^2 + \text{Odd}_{n+1} \quad (\text{Using the induction hypothesis}) \\ &= n^2 + 2n + 1 \quad (\text{Odd}_{n+1} \text{ is } 2n + 1) \\ &= (n+1)^2 \end{aligned}$$

Thus we have shown that the sum of the first  $n+1$  odd integers must be equivalent to  $(n+1)^2$  if it is known that the sum of the first  $n$  of them is equivalent to  $n^2$ .

Mathematical induction lets us prove properties of positive integers. But it also lets us prove properties of other things if the properties are described in terms of integers. For example, we could talk about the size of finite sets, or the length of finite strings. Let's do one with sets: For any finite set  $A$ ,  $|2^A| = 2^{|A|}$ . In other words, the cardinality of the power set of  $A$  is 2 raised to the power of the cardinality of  $A$ . We'll prove this by induction on the number of elements of  $A$  ( $|A|$ ). We follow the same two steps:

1. Let  $v = 0$ . So  $A$  is  $\emptyset$ ,  $|A| = 0$ , and  $A$ 's power set is  $\{\emptyset\}$ , whose cardinality is  $1 = 2^0 = 2^{|A|}$ .
2. Prove that,  $\forall n \geq 0$ , if  $|2^A| = 2^{|A|}$  is true for all sets  $A$  of cardinality  $n$ , then it is also true for all sets  $S$  of cardinality  $n+1$ .

We do this as follows. Since  $n \geq 0$ , and any such  $S$  has  $n + 1$  elements,  $S$  must have at least one element. Pick one and call it  $a$ . Now consider the set  $T$  that we get by removing  $a$  from  $S$ .  $|T|$  must be  $n$ . So, by the induction hypothesis (namely that  $|2^A| = 2^{|A|}$  if  $|A| = n$ ), our claim is true for  $T$  and we have  $|2^T| = 2^{|T|}$ . Now let's return to the power set of the original set  $S$ . It has two parts: those subsets that include  $a$  and those that don't. The second part is exactly  $2^T$ , so we know that it has  $2^{|T|} = 2^n$  elements. The first part (all the subsets that include  $a$ ) is exactly all the subsets that don't include  $a$  with  $a$  added in. Since there are  $2^n$  subsets that don't include  $a$  and there are the same number of them once we add  $a$  to each, we have that the total number of subsets of our original set  $S$  is  $2^n$  (for the ones that don't include  $a$ ) plus  $2^n$  (for the ones that do include  $a$ ), for a total of  $2(2^n) = 2^{n+1}$ , which is exactly  $2^{|S|}$ .

Why does mathematical induction work? It relies on the **well-ordering property** of the integers, which states that every nonempty set of nonnegative integers has a least element. Let's see how that property assures us that mathematical induction is valid as a proof technique. We'll use the technique of proof by contradiction to show that, given the well-ordering property, mathematical induction must be valid. Once we have done an induction proof, we know that  $A(v)$  (where  $v$  is 0 or 1 or some other starting value) is true and we know that  $\forall n \geq 0$ ,  $A(n) \Rightarrow A(n+1)$ . What we're using the technique to enable us to claim is that, therefore,  $\forall n \geq v$ ,  $A(n)$ . Suppose the technique were not valid and there was a set  $S$  of nonnegative integers  $\geq v$  for which  $A(n)$  is False. Then, by the well-ordering property, there is a smallest element in this set. Call it  $x$ . By definition,  $x$  must be equal to or greater than  $v$ . But it cannot actually be  $v$  because we proved  $A(v)$ . So it must be greater than  $v$ . But now consider  $x - 1$ . Since  $x - 1$  is less than  $x$ , it cannot be in  $S$  (since we chose  $x$  to be the smallest value in  $S$ ). If it's not in  $S$ , then we know  $A(x - 1)$ . But we proved that  $\forall n \geq 0$ ,  $A(n) \Rightarrow A(n+1)$ , so  $A(x - 1) \Rightarrow A(x)$ . But we assumed  $\neg A(x)$ . So that assumption led us to a contradiction; thus it must be false.

Sometimes the principle of mathematical induction is stated in a slightly different but formally equivalent way:

1. Prove that  $A$  holds for the smallest value  $v$  with which we're concerned.
2. State the **induction hypothesis**  $H$ , which must be of the form, "There is some integer  $n \geq v$  such that  $A$  is true for all integers  $k$  where  $v \leq k \leq n$ ."
3. Prove that  $(\forall n \geq v, A(n)) \Rightarrow A(n+1)$ . In other words prove that whenever  $A$  holds for all nonnegative integers starting with  $v$ , up to an including  $n$ , it must also hold for  $n+1$ .

You can use whichever form of the technique is easiest for a particular problem.



# Regular Languages and Finite State Machines

## 1 Regular Languages

The first class of languages that we will consider is the regular languages. As we'll see later, there are several quite different ways in which regular languages can be defined, but we'll start with one simple technique, regular expressions.

A **regular expression** is an expression (string) whose "value" is a language. Just as  $3 + 4$  and  $14/2$  are two arithmetic expressions whose values are equal, so are  $(a \cup b)^*$  and  $a^* \cup (a \cup b)^*b(a \cup b)^*$  two regular expressions whose values are equal. We will use regular expressions to denote languages just as we use arithmetic expressions to denote numbers. Just as there are some numbers, like  $\pi$ , that cannot be expressed by arithmetic expressions of integers, so too there are some languages, like  $a^n b^n$ , that cannot be expressed by regular expressions. In fact, we will define the class of **regular languages** to be precisely those that *can* be described with regular expressions.

Let's continue with the analogy between arithmetic expressions and regular expressions. The syntax of arithmetic expressions is defined recursively:

1. Any numeral in  $\{0, 1, 2, \dots\}$  is an arithmetic expression.
2. If  $\alpha$  and  $\beta$  are expressions, so is  $(\alpha + \beta)$ .
3. If  $\alpha$  and  $\beta$  are expressions, so is  $(\alpha * \beta)$ .
4. If  $\alpha$  and  $\beta$  are expressions, so is  $(\alpha - \beta)$ .
5. If  $\alpha$  and  $\beta$  are expressions, so is  $(\alpha/\beta)$ .
6. If  $\alpha$  is an expression, so is  $-\alpha$ .

These operators that we've just defined have associated with them a set of precedence rules, so we can write  $-3 + 4*5$  instead of  $(-3 + (4*5))$ .

Now let's return to regular expressions. The syntax of regular expressions is also defined recursively:

1.  $\emptyset$  and each member of  $\Sigma$  is a regular expression.
2. If  $\alpha, \beta$  are regular expressions, then so is  $\alpha\beta$ .
3. If  $\alpha, \beta$  are regular expressions, then so is  $\alpha \cup \beta$ .
4. If  $\alpha$  is a regular expression, then so is  $\alpha^*$ .
5. If  $\alpha$  is a regular expression, then so is  $(\alpha)$ .
6. Nothing else is a regular expression.

Similarly there are precedence rules for regular expressions, so we can write  $a^* \cup bc$  instead of  $(a^* \cup (bc))$ . Note that  $*$  binds more tightly than does concatenation, which binds more tightly than  $\cup$ .

In both cases (arithmetic expressions and regular expressions) there is a distinction between the expression and its value.  $5 + 7$  is not the same expression as  $3 * 4$ , even though they both have the same value. (You might imagine the expressions being in quotation marks: " $5 + 7$ " is clearly not the same as " $3 * 4$ ". Similarly, "John's a good guy" is a different sentence from "John is nice", even though they both have the same meaning, more or less.) The rules that determine the value of an arithmetic expression are quite familiar to us, so much so that we are usually not consciously aware of them. But regular expressions are less familiar, so we will explicitly specify the rules that define the values of regular expressions. We do this by recursion on the structure of the expression. Just remember that the regular expression itself is a syntactic object made of parentheses and other symbols, whereas its value is a language. We define  $L()$  to be the function that maps regular expressions to their values. We might analogously define  $L()$  for arithmetic expressions and say that  $L(5 + 7) = 12 = L(3 * 4)$ .  $L()$  is an example of a **semantic interpretation function**, i. e., it is a function that maps a string in some language to its meaning. In our case, of course, the language from which the arguments to  $L()$  will be drawn is the language of regular expressions as defined above.

$L()$  for regular expressions is defined as follows:

1.  $L(\emptyset) = \emptyset$  and  $L(a) = \{a\}$  for each  $a \in \Sigma$
2. If  $\alpha, \beta$  are regular expressions, then

$$L((\alpha\beta)) = L(\alpha) L(\beta)$$

= all strings that can be formed by concatenating to some string from  $\alpha$  some string from  $\beta$ .

Note that if either  $\alpha$  or  $\beta$  is  $\emptyset$ , then its language is  $\emptyset$ , so there is nothing to concatenate and the result is  $\emptyset$ .

3. If  $\alpha, \beta$  are regular expressions, then  $L((\alpha\cup\beta)) = L(\alpha) \cup L(\beta)$
4. If  $\alpha$  is a regular expression, then  $L(\alpha^*) = L(\alpha)^*$
5.  $L(\epsilon) = L(\alpha)$

So, for example, let's find the meaning of the regular expression  $(a \cup b)^*b$ :

$$\begin{aligned} L((a \cup b)^*b) &= L((a \cup b)^*) L(b) \\ &= L(a \cup b)^* L(b) \\ &= (L(a) \cup L(b))^* L(b) \\ &= (\{a\} \cup \{b\})^* \{b\} \\ &= \{a, b\}^* \{b\} \end{aligned}$$

which is just the set of all strings ending in  $b$ . Another example is  $L(((a \cup b)(a \cup b)a(a \cup b)^*)) = \{xay: x \text{ and } y \text{ are strings of } a\text{'s and } b\text{'s and } |x| = 2\}$ . The distinction between an expression and its meaning is somewhat pedantic, but you should try to understand it. We will usually not actually write  $L()$  because it is generally clear from context whether we mean the regular expression or the language denoted by it. For example,  $a \in (a \cup b)^*$  is technically meaningless since  $(a \cup b)^*$  is a regular expression, not a set. Nonetheless, we use it as a reasonable abbreviation for  $a \in L((a \cup b)^*)$ , just as we write  $3 + 4 = 4 + 3$  to mean that the values of "3 + 4" and "4 + 3", not the expressions themselves, are identical.

Here are some useful facts about regular expressions and the languages they describe:

- $(a \cup b)^* = (a^*b^*)^* = (b^*a^*)^*$  = set of all strings composed exclusively of  $a$ 's and  $b$ 's (including the empty string)
- $(a \cup b)c = (ac \cup bc)$  Concatenation distributes over union
- $c(a \cup b) = (ca \cup cb)$  "
- $a^* \cup b^* \neq (a \cup b)^*$  The right-hand expression denotes a set containing strings of mixed  $a$ 's and  $b$ 's, while the left-hand expression does not.
- $(ab)^* \neq a^*b^*$  In the right-hand expression, all  $a$ 's must precede all  $b$ 's. That's not true for the left-hand expression.
- $a^* \cup \emptyset^* = a^* \cup \epsilon = a^*$

There is an algebra of regular expressions, but it is rather complex and not worth the effort to learn it. Therefore, we will rely primarily on our knowledge of what the expressions mean to determine the equivalence (or non-equivalence) of regular expressions.

We are now in a position to state formally our definition of the class of **regular languages**: Given an alphabet  $\Sigma$ , the set of regular languages over  $\Sigma$  is precisely the set of languages that can be defined using regular expressions with respect to  $\Sigma$ . Another equivalent definition (given our definition of regular expressions and  $L()$ ), is that the set of regular languages over an alphabet  $\Sigma$  is the smallest set that contains  $\emptyset$  and each of the elements of  $\Sigma$ , and that is closed under the operations of concatenation, union, and Kleene star (rules 2, 3, and 4 above).

## 2 Proof of the Equivalence of Nondeterministic and Deterministic FSAs

In the lecture notes, we stated the following:

**Theorem:** For each NDFSA, there is an equivalent DFSA.

This is an extremely significant theorem. It says that, for finite state automata, nondeterminism doesn't add power. It adds convenience, but not power. As we'll see later, this is not true for all other classes of automata.

In the notes, we provided the first step of a proof of this theorem, namely an algorithm to construct, from any NDFSA, an equivalent DFSA. Recall that the algorithm we presented was the following: Given a nondeterministic FSA  $M = (K, \Sigma, \Delta, s, F)$ , we derive an equivalent deterministic FSA  $M' = (K', \Sigma, \delta', s', F')$  as follows:

1. Compute  $E(q)$  for each  $q$  in  $K$ .  $\forall q \in K, E(q) = \{p \in K : (q, \epsilon) \vdash_M^* (p, \epsilon)\}$ . In other words,  $E(q)$  is the set of states reachable from  $q$  without consuming any input.
2. Compute  $s' = E(s)$ .
3. Compute  $\delta'$ , which is defined as follows:  $\forall Q \subseteq 2^K$  and  $\forall a \in \Sigma, \delta'(Q, a) = \cup \{E(p) : p \in K \text{ and } (q, a, p) \in \Delta \text{ for some } q \in Q\}$ . Recall that the elements of  $2^K$  are sets of states from the original machine  $M$ . So what we've just said is that to compute the transition out of one of these "set" states, find all the transitions out of the component states in the original machine, then find all the states reachable from them via epsilon transitions. The new state is the set of all states reachable in this way. We'll actually compute  $\delta'$  by first computing it for the new start state  $s'$  and each of the elements of  $\Sigma$ . Each state thus created becomes an element of  $K'$ , the set of states of  $M'$ . Then we compute  $\delta'$  for any new states that were just created. We continue until there are no additional reachable states. (so although  $\delta'$  is defined for all possible subsets of  $K$ , we'll only bother to compute it for the reachable such subsets and in fact we'll define  $K'$  to include just the reachable configurations.)
4. Compute  $K' =$  that subset of  $2^K$  that is reachable, via  $\delta'$ , as defined in step 3, from  $s'$ .
5. Compute  $F' = \{Q \in K' : Q \cap F \neq \emptyset\}$ . In other words, each constructed "set" state that contains at least one final state from the original machine  $M$  becomes a final state in  $M'$ .

However, to complete the proof of the theorem that asserts that there is an *equivalent* DFSA for every NDFSA, we need next to prove that the algorithm we have defined does in fact produce a machine that is (1) deterministic, and (2) equivalent to the original machine.

Proving (1) is trivial. By the definition in step 3 of  $\delta'$ , we are guaranteed that  $\delta'$  is defined for all reachable elements of  $K'$  and that it is single valued.

Next we must prove (2). In other words, we must prove that  $M'$  accepts a string  $w$  if and only if  $M$  accepts  $w$ . We constructed the transition function  $\delta'$  of  $M'$  so that each step of the operation of  $M'$  mimics an "all paths" simulation of  $M$ . So we believe that the two machines are identical, but how can we prove that they are? Suppose we could prove the following:

**Lemma:** For any string  $w \in \Sigma^*$  and any states  $p, q \in K$ ,  $(q, w) \vdash_M^* (p, \epsilon)$  iff  $(E(q), w) \vdash_{M'}^* (P, \epsilon)$  for some  $P \in K'$  that contains  $p$ . In other words, if the original machine  $M$  starts in state  $q$  and, after reading the string  $w$ , can land in state  $p$ , then the new machine  $M'$  must behave as follows: when started in the state that corresponds to the set of states the original machine  $M$  could get to from  $q$  without consuming any input,  $M'$  reads the string  $w$  and lands in one of its new "set" states that contains  $p$ . Furthermore, because of the only-if part of the lemma,  $M'$  must end up in a "set" state that contains only states that  $M$  could get to from  $q$  after reading  $w$  and following any available epsilon transitions.

If we assume that this lemma is true, then the proof that  $M'$  is equivalent to  $M$  is straightforward: Consider any string  $w \in \Sigma^*$ . If  $w \in L(M)$  (i.e., the original machine  $M$  accepts  $w$ ) then the following two statements must be true:

1. The original machine  $M$ , when started in its start state, can consume  $w$  and end up in a final state. This must be true given the definition of what it means for a machine to accept a string.

2.  $(E(s), w) \vdash_{M'}^* (Q, \epsilon)$  for some  $Q$  containing some  $f \in F$ . In other words, the new machine, when started in its start state, can consume  $w$  and end up in one of its final states. This follows from the lemma, which is more general and describes a computation from any state to any other. But if we use the lemma and let  $q$  equal  $s$  (i.e.,  $M$  begins in its start state) and  $p = f$  for some  $f \in F$  (i.e.,  $M$  ends in a final state), then we have that the new machine  $M'$ , when started in its start state,  $E(s)$ , will consume  $w$  and end in a state that contains  $f$ . But if  $M'$  does that, then it has ended up in one of its final states (by the definition of  $K'$  in step 5 of the algorithm). So  $M'$  accepts  $w$  (by the definition of what it means for a machine to accept a string). Thus  $M'$  accepts precisely the same set of strings that  $M$  does.

Now all we have to do is to prove the lemma. What the lemma is saying is that we've built  $M'$  from  $M$  in such a way that the computations of  $M'$  mirror those of  $M$  and guarantee that the two machines accept the same strings. But of course we didn't build  $M'$  to perform an entire computation. All we did was to describe how to construct  $\delta'$ . In other words, we defined how individual steps of the computation work. What we need to do now is to show that the individual steps, when taken together, do the right thing for strings of any length. The obvious way to do that, since we know what happens one step at a time, is to prove the lemma by induction on  $|w|$ .

We must first prove that the lemma is true for the base case, where  $|w| = 0$  (i.e.,  $w = \epsilon$ ). To do this, we actually have to do two proofs, one to establish the *if* part of the lemma, and the other to establish the *only if* part:

Basis step, if part: Prove  $(q, w) \vdash_{M'}^* (p, \epsilon)$  if  $(E(q), w) \vdash_M^* (P, \epsilon)$  for some  $P \in K'$  that contains  $p$ . Or, turning it around to make it a little clearer,

$$[(E(q), w) \vdash_M^* (P, \epsilon) \text{ for some } P \in K' \text{ that contains } p] \Rightarrow (q, w) \vdash_{M'}^* (p, \epsilon)$$

If  $|w| = 0$ , then  $M'$  makes no moves. So it must end in the same state it started in, namely  $E(q)$ . If we're told that it ends in some state that contains  $p$ , then  $p \in E(q)$ . But, given our definition of  $E(x)$ , that means exactly that, in the original machine  $M$ ,  $p$  is reachable from  $q$  just by following  $\epsilon$  transitions, which is exactly what we need to show.

Basis step, only if part: Recall that *only if* is equivalent to *implies*. So now we need to show:

$$[(q, w) \vdash_{M'}^* (p, \epsilon)] \Rightarrow (E(q), w) \vdash_M^* (P, \epsilon) \text{ for some } P \in K' \text{ that contains } p$$

If  $|w| = 0$ , and the original machine  $M$  goes from  $q$  to  $p$  with only  $w$  as input, it must go from  $q$  to  $p$  following just  $\epsilon$  transitions. In other words  $p \in E(q)$ . Now consider the new machine  $M'$ . It starts in  $E(q)$ , the set state that includes all the states that are reachable from  $q$  via  $\epsilon$  transitions. Since the new machine is deterministic, it will make no moves at all if its input is  $\epsilon$ . So it will halt in exactly the same state it started in, namely  $E(q)$ . Since we know that  $p \in E(q)$ , we know that  $M'$  has halted in a set state that includes  $p$ , which is exactly what we needed to show.

Next we'll prove that if the lemma is true for all strings  $w$  of length  $k$ ,  $k \geq 0$ , then it is true for all strings of length  $k + 1$ . Considering strings of length  $k + 1$ , we know that we are dealing with strings of a least one character. So we can rewrite any such string as  $zx$ , where  $x$  is a single character and  $z$  is a string of length  $k$ . The way that  $M$  and  $M'$  process  $z$  will thus be covered by the induction hypothesis. We'll use our definition of  $\delta'$ , which specifies how each individual step of  $M'$  operates, to show that, assuming that the machines behave correctly for the first  $k$  characters, they behave correctly for the last character also and thus for the entire string of length  $k + 1$ . Recall our definition of  $\delta'$ :

$$\delta'(Q, a) = \cup \{E(p) : p \in K \text{ and } (q, a, p) \in \Delta \text{ for some } q \in Q\}.$$

To prove the lemma, we must show a relationship between the behavior of:

$M$ :  $(q, w) \vdash_M^* (p, \epsilon)$ , and

$M'$ :  $(E(q), w) \vdash_{M'}^* (P, \epsilon)$  for some  $P \in K'$  that contains  $p$

Rewriting  $w$  as  $zx$ , we have

$M$ :  $(q, zx) \vdash_M^* (p, \epsilon)$

$M'$ :  $(E(q), zx) \vdash_{M'}^* (P, \epsilon)$  for some  $P \in K'$  that contains  $p$

Breaking each of these computations into two pieces, the processing of  $z$  followed by the processing of  $x$ , we have:

$M$ :  $(q, zx) \vdash_M^* (s_i, x) \vdash_M^* (p, \epsilon)$

$M'$ :  $(E(q), zx) \vdash_{M'}^* (S, x) \vdash_{M'}^* (P, \epsilon)$  for some  $P \in K'$  that contains  $p$

In other words, after processing  $z$ ,  $M$  will be in some set of states  $s_i$ , and  $M'$  will be in some state, which we'll call  $S$ . Again, we'll split the proof into two parts:

Induction step, *if* part:

$$[(E(q), zx) \vdash_{M'}^* (S, x) \vdash_{M'}^* (P, \epsilon) \text{ for some } P \in K' \text{ that contains } p] \Rightarrow (q, zx) \vdash_M^* (s_i, x) \vdash_M^* (p, \epsilon)$$

If, after reading  $z$ ,  $M'$  is in state  $S$ , we know, from the induction hypothesis, that the original machine  $M$ , after reading  $z$ , must be in some set of states  $s_i$  and that  $S$  is precisely that set. Now we just have to describe what happens at the last step when the two machines read  $x$ . If we have that  $M'$ , starting in  $S$  and reading  $x$  lands in  $P$ , then we know, from the definition of  $\delta'$  above, that  $P$  contains precisely the states that  $M$  could land in after starting in any  $s_i$  and reading  $x$ . Thus if  $p \in P$ ,  $p$  must be a state that  $M$  could land in.

Induction step, *only if* part:

$$(q, zx) \vdash_M^* (s_i, x) \vdash_M^* (p, \epsilon) \Rightarrow (E(q), zx) \vdash_{M'}^* (S, x) \vdash_{M'}^* (P, \epsilon) \text{ for some } P \in K' \text{ that contains } p$$

By the induction hypothesis, we know that if  $M$ , after processing  $z$ , can reach some set of states  $s_i$ , then  $S$  (the state  $M'$  is in after processing  $z$ ) must contain precisely all the  $s_i$ 's. Knowing that, and our definition of  $\delta'$ , we know that from  $S$ , reading  $x$ ,  $M'$  must be in some set state  $P$  that contains precisely the states that  $M$  can reach starting in any of the  $s_i$ 's, reading  $x$ , and then following all  $\epsilon$  transitions. So, after consuming  $w(zx)$ ,  $M'$ , when started in  $E(q)$  must end up in a state  $P$  that contains all and only the states  $p$  that  $M$ , when started in  $q$ , could end up in.

This theorem is a very useful result when we're dealing with FSAs. It's true that, by and large, we want to build deterministic machines. But, because we have provided a constructive proof of this theorem, we know that we can design a deterministic FSA by first describing a nondeterministic one (which is sometimes a much simpler task), and then applying our algorithm to construct a corresponding deterministic one.

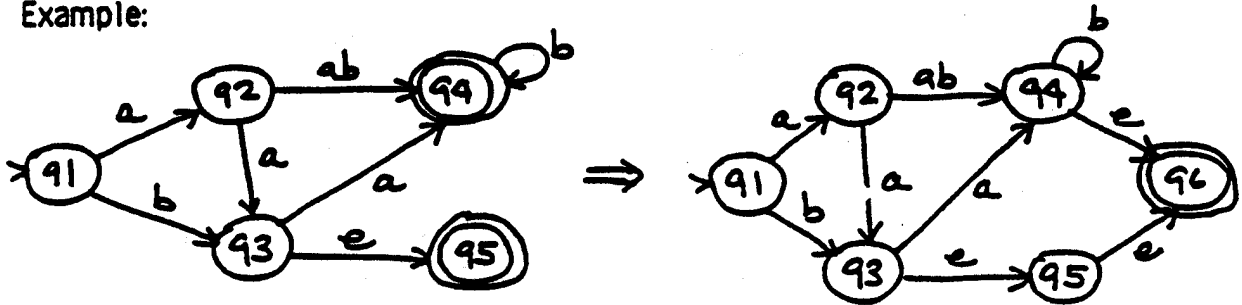
### 3 Generating Regular Expressions from Finite State Machines

#### I. Preparations (Note: FA may be non-deterministic in general)

If a) there is more than one final state  
or b) there is just one final state but it lies on a loop,

then a) create a new final state  
b) run e-transitions from the old final state(s) to the new one  
and c) make the old final state(s) non-final

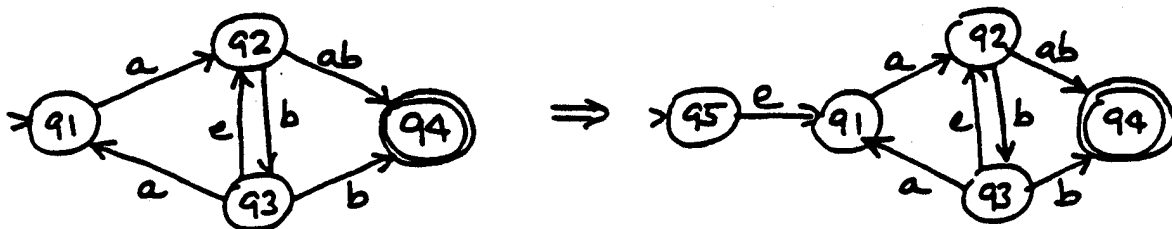
Example:



If the initial state is part of a loop,

then a) create a new initial state  
b) run an e-transition from the new initial state to the old one  
and c) make the old initial state non-initial

Example:



(Note: nothing needs to be done to the final state here because it does not lie on a loop. Similarly, nothing needs to be done to the initial state in the first example.)

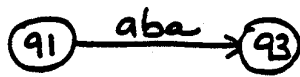
## II. Eliminating states

One by one, remove states which are neither initial nor final, replacing the connections between remaining states in such a way that the transitions are preserved. In general, the reconstructed transitions may be labelled with regular expressions rather than just by strings.

### 1. Example:



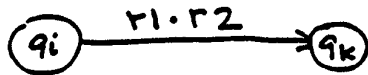
q2 can be eliminated and the connection between q1 and q3 becomes:



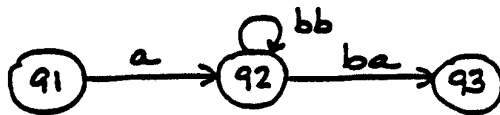
In general, if  $r_1$  and  $r_2$  are any regular expressions, produced perhaps by other steps in the algorithm, and occur in the configuration:



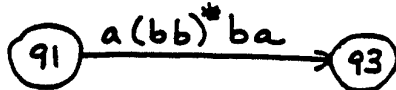
then this can be replaced by



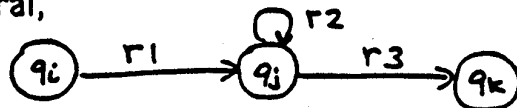
2) If the eliminated state happens to contain a "simple" loop, e.g.:



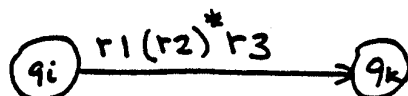
when q2 is eliminated, the transition becomes:



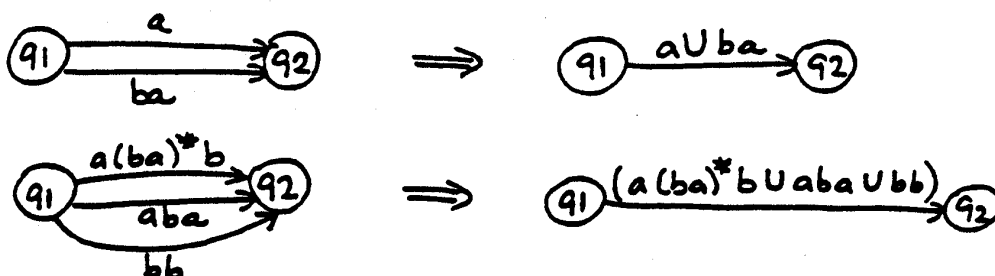
In general,



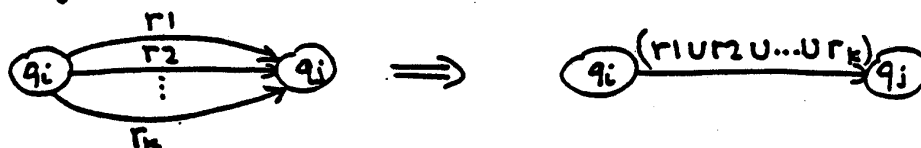
becomes



3) Parallel transitions can be coalesced into a single transition labelled by an expression which is the union of the originals:



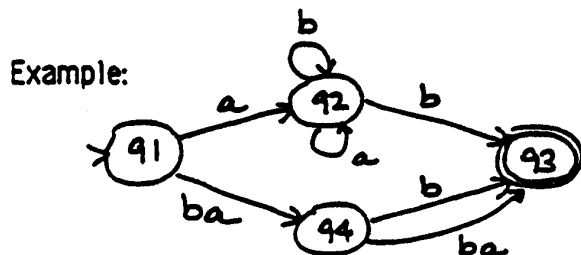
In general:



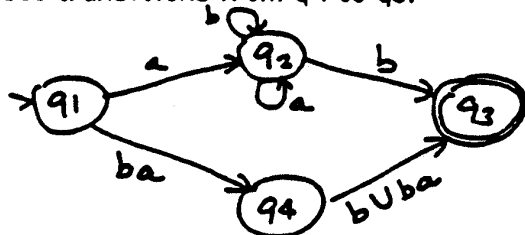
One proceeds in this manner, eliminating states one by one, until only a single transition remains connecting the initial and the one final state. The label on this transition is a regular expression for the original automaton.

The order of elimination of states doesn't matter; equivalent regular expressions will be obtained so long as the procedure is done correctly.

Note that coalescing parallel transitions doesn't eliminate a state but reduces the number of transitions. In general, one should perform this step before eliminating a state that has parallel transitions into or out of it.

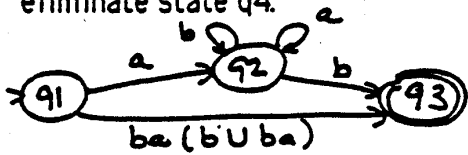


coalesce transitions from q4 to q3:

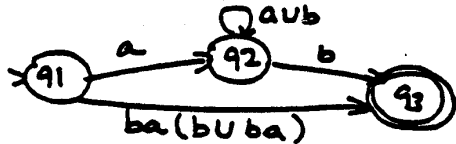




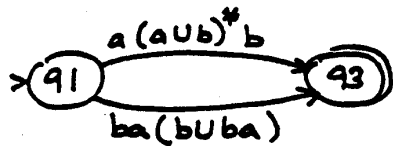
eliminate state q4:



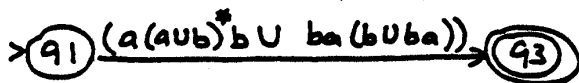
coalesce transitions from q2 to q2:



eliminate q2:

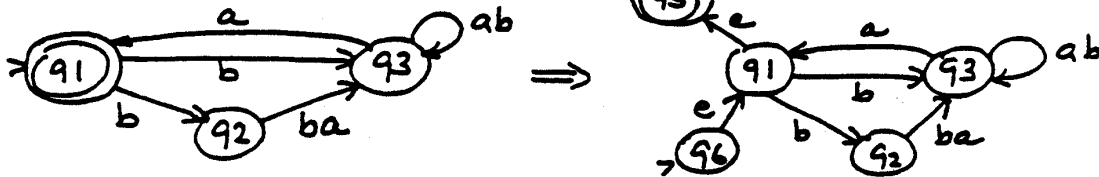


coalesce parallel transitions:

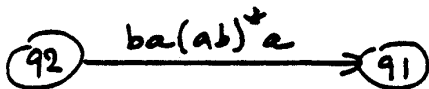


This is a regular expression for the FA. Check against the original.

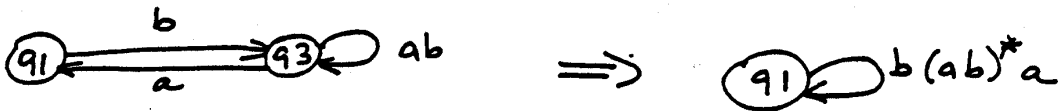
More complicated cases:



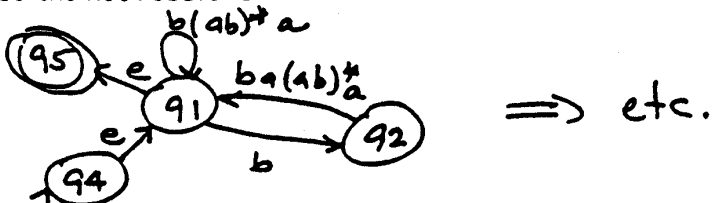
suppose we eliminate q3. Then the path from q2 to q1 becomes:



but we have also destroyed a path from q1 back to q1, namely:



so the net result is:



## 4 The Pumping Lemma for Regular Languages

### 4.1 What Does the Pumping Lemma Tell Us?

The pumping lemma is a powerful technique for showing that a language is **not** regular. The lemma describes a property that must be true of any language if it is regular. Thus, if we can show that some language  $L$  does not possess this property, then we know that  $L$  is not regular.

The key idea behind the pumping lemma derives from the fact that every regular language is recognizable by some FSM  $M$  (actually an infinite number of finite state machines, but we can just pick any one of them for our purposes here). So we know that, if a language  $L$  is regular, then every string  $s$  in  $L$  must drive  $M$  from the start state to some final state. There are only two ways this can happen:

1.  $s$  could be short and thus drive  $M$  from the start state to a final state without traversing any loops. By short we mean that if  $M$  has  $N$  states, then  $|s| < N$ . If  $s$  were any longer,  $M$  would have to visit some state more than once while processing  $s$ ; in other words it would loop.
2.  $s$  could be long and  $M$  could traverse at least one loop. If it does that, then observe that another way to take  $M$  from the start state to the same final state would be to skip the loop. Some shorter string  $w$  in  $L$  would do exactly that. Still further ways to take  $M$  from the start state to the final state would be to traverse the loop two times, or three times, or any number of times. An infinite set of longer strings in  $L$  would do this.

Given some regular language  $L$ , we know that there exists an infinite number of FSMs that accept  $L$ , and the pumping lemma relies on the existence of at least one such machine  $M$ . In fact, it needs to know the number of states in  $M$ . But what if we don't have  $M$  handy? No problem. All we need is to know that  $M$  exists and that it has some number of states, which we'll call  $N$ . As long as we make no assumptions about what  $N$  is, other than that it is at least 1, we can forge ahead without figuring out what  $M$  is or what  $N$  is.

The pumping lemma tells us that if a language  $L$  is regular, then any sufficiently long (as defined above) string  $s$  in  $L$  must be "pumpable". In other words, there is some substring  $t$  of  $s$  that corresponds to a loop in the recognizing machine  $M$ . Any string that can be derived from  $s$  either by pumping  $t$  out once (to get a shorter string that will go through the loop one fewer times) or by pumping  $t$  in any number of times (to get longer strings that will go through the loop additional times) must also be in  $L$  since they will drive  $M$  from its start state to its final state. If we can show that there is even one sufficiently long string  $s$  that isn't pumpable, then we know that  $L$  must not be regular.

You may be wondering why we can only guarantee that we can pump out once, yet we must be able to pump in an arbitrary number of times. Clearly we must be able to pump in an arbitrary number of times. If there's a loop in our machine  $M$ , there is no limit to the number of times we can traverse it and still get to a final state. But why only once for pumping out? Sometimes it may in fact be possible to pump out more. But the lemma doesn't require that we be able to. Why? When we pick a string  $s$  that is "sufficiently long", all we know is that it is long enough that  $M$  must visit at least one state more than once. In other words, it must traverse at least one loop of length one at least once. It may do more, but we can't be sure of it. So the only thing we're guaranteed is that we can pump out the one pass through the loop that we're sure must exist.

#### ***Pumping Lemma:***

If  $L$  is regular, then

$\exists N \geq 1$ , such that

$\forall$  strings  $w$ , where  $|w| \geq N$ ,

$\exists x, y, z$ , such that  $w = xyz$ , and

$|xy| \leq N$ , and

$y \neq \epsilon$ , and

$\forall q \geq 0$ ,  $xy^qz$  is in  $L$ .

The lemma we've just stated is sometimes referred to as the Strong Pumping Lemma. That's because there is a weaker version that is much less easy to use, yet no easier to prove. We won't say anything more about it, but at least now you know what it means if someone refers to the Strong Pumping Lemma.

## 4.2 Using the Pumping Lemma

The key to using the pumping lemma correctly to prove that a language L is not regular is to understand the nested quantifiers in the lemma. Remember, our goal is to show that our language L fails to satisfy the requirements of the lemma (and thus is not regular). In other words, we're looking for a counterexample. But when do we get to pick any example we want and when do we have to show that there is no example? The lemma contains both universal and existential quantifiers, so we'd expect some of each. But the key is that we want to show that the lemma does not apply to our language L. So we're essentially taking the not of it. What happens when we do that? Remember two key results from logic:

$$\neg \forall x P(x) = \exists x \neg P(x)$$

$$\neg \exists x P(x) = \forall x \neg P(x)$$

So if the lemma says something must be true for all strings, we show that there exists at least one string for which it's false. If the lemma says that there exists some object with some properties, we show that all possible objects fail to have those properties. More specifically:

At the top level, the pumping lemma states

$$L \text{ regular} \Rightarrow \exists N \geq 1, P(L, N), \text{ where } P \text{ is the rest of the lemma (think of } P \text{ as the Pumpable property).}$$

To show that the lemma does not correctly describe our language L, we must show

$$\neg(\exists N \geq 1, P(L, N)), \text{ or, equivalently,}$$

$$\forall N \neg P(L, N)$$

The lemma asserts first that there exists some magic number N, which defines what we mean by "sufficiently long." The lemma doesn't tell us what N is (although we know it derives from the number of states in some machine M that accepts L). We need to show that no such N exists. So we don't get to pick a specific N to work with. Instead:

**We must carry N through the rest of what we do as a variable and make no assumptions about it.**

Next, we must look inside the pumpable property. The lemma states that every string that is longer than N must be pumpable. To prove that that isn't true of L, all we have to do is to find a single long string in L that isn't pumpable. So:

**We get to pick a string w.**

Next, the lemma asserts that there is a way to carve up our string into substrings x, y, and z such that pumping works. So we must show that there is no such x, y, z triple. This is the tricky part. To show this, we must enumerate all logically possible ways of carving up w into x, y, and z. For each such possibility, we must show that at least one of the pumping requirements is not satisfied. So:

**We don't get to pick x, y, and z. We must show that pumping fails for all possible x, y, z triples.**

Sometimes, we can show this easily without actually enumerating ways of carving up w into x, y, and z. But in other cases, it may be necessary to enumerate two or more possibilities.

Let's look at an example. The classic one is  $L = a^x b^x$  is not regular. Intuitively, we knew it couldn't be. To decide whether a string is in L, we have to count the a's, and then compare that number to the number of b's. Clearly no machine with a finite number of states can do that. Now we're actually in a position to prove this. We show that for any value of N we'll get a contradiction. We get to choose any w we want as long as its length is greater than or equal to N. Let's choose w to be  $a^N b^N$ . Next, we must show that there is no x, y, z triple with the required properties:

$$|xy| \leq N,$$

$$y \neq \epsilon,$$

$$\forall q \geq 0, xy^qz \text{ is in } L.$$

Suppose there were. Then it might look something like this (this is just for illustration):

$$\begin{array}{c} 1 \quad | \quad 2 \\ \hline a a a a a a a a b b b b b b b b \end{array}$$

x      y      z

Don't take this picture to be making any claim about what  $x$ ,  $y$ , and  $z$  are. But what the picture does show is that  $w$  is composed of two regions:

1. The initial segment, which is all a's.
2. The final segment, which is all b's.

Typically, as we attempt to show that there is no  $x, y, z$  triple that satisfies all the conditions of the pumping lemma, what we'll do is to consider the ways that  $y$  can be spread within the regions of  $w$ . In this example, we observe immediately that since  $|xy| \leq N$ ,  $y$  must be  $a^g$  for some  $g \geq 1$ . (In other words,  $y$  must lie completely within the first region.) Now there's just one case to consider. Clearly we'll add just a's as we pump, so there will be more a's than b's, so we'll generate strings that are not in  $L$ . Thus  $w$  is not pumpable, we've found a contradiction, and  $L$  is not regular.

The most common mistake people make in applying the pumping lemma is to show a particular  $x, y, z$  triple that isn't pumpable. Remember, you must show that all such triples fail to be pumpable.

Suppose you try to apply the pumping lemma to a language  $L$  and you fail to find a counterexample. In other words, every string  $w$  that you examine is pumpable. What does that mean? Does it mean that  $L$  is regular? No. It's true that if  $L$  is regular, you will be unable to find a counterexample. But if  $L$  isn't regular, you may fail if you just don't find the right  $w$ . In other words, even in a non regular language, there may be plenty of strings that are pumpable. For example, consider  $L = a^x b^x \cup a^y$ . In other words, if there are any b's there must be the same number of them as there are a's, but it's also okay just to have a's. We can prove that this language is not pumpable by choosing  $w = a^N b^N$ , just as we did for our previous example, and the proof will work just as it did above. But suppose that were less clever. Let's choose  $w = a^N$ . Now again we know that  $y$  must be  $a^g$  for some  $g \geq 1$ . But now, if we pump  $y$  either in or out, we still get strings in  $L$ , since all strings that just contain a's are in  $L$ . We haven't proved anything.

Remember that, when you go to apply the pumping lemma, the one thing that is in your control is the choice of  $w$ . As you get experience with this, you'll notice a few useful heuristics that will help you find a  $w$  that is easy to work with:

1. Choose  $w$  so that there are distinct regions, each of which contains only one element of  $\Sigma$ . When we considered  $L = a^x b^x$ , we had no choice about this, since every element of  $L$  (except  $\epsilon$ ) must have a region of a's followed by a region of b's. But suppose we were interested in  $L' = \{w \in \{a, b\}^* : w \text{ contains an equal number of a's and b's}\}$ . We might consider choosing  $w = (ab)^N$ . But now there are not clear cut regions. We won't be able to use pumping successfully because if  $y = ab$ , then we can pump to our hearts delight and we'll keep getting strings in  $L$ . What we need to do is to choose  $a^N b^N$ , just as we did when we were working on  $L$ . Sure,  $L'$  doesn't require that all the a's come first. But strings in which all the a's do come first are fine elements of  $L'$ , and they produce clear cut regions that make the pumping lemma useful.
2. Choose  $w$  so that the regions are big enough that there is a minimal number of configurations for  $y$  across the regions. In particular, you must pick  $w$  so that it has length at least  $N$ . But there's no reason to be parsimonious. For example, when we were working on  $a^x b^x$ , we could have chosen  $w = a^{N/2} b^{N/2}$ . That would have been long enough. But then we couldn't have known that  $y$  would be a string of a's. We would have had to consider several different possibilities for  $y$ . (You might want to work this one out to see what happens.) It will generally help to choose  $w$  so that each region is of length at least  $N$ .
3. Whenever possible, choose  $w$  so that there are at least two regions with a clear boundary between them. In particular, you want to choose  $w$  so that there are at least two regions that must be related in some way (e.g., the a region must be the same length as the b region). If you follow this rule and rule 2 at the same time, then you'll be assured that as you pump  $y$ , you'll change one of the regions without changing the other, thus producing strings that aren't in your language.

The pumping lemma is a very powerful tool for showing that a language isn't regular. But it does take practice to use it right. As you're doing the homework problems for this section, you may find it helpful to use the worksheet that appears on the next page.

# Using the Pumping Lemma for Regular Languages

If  $L$  is regular, then

There exists an  $N \geq 1$ , (Just **call it  $N$** ) such that

for all strings  $w$ , where  $|w| \geq N$ ,

(Since true for all  $w$ , it must be true for any particular one, so you **pick  $w$** )

(Hint: describe  $w$  in terms of  $N$ )

there exist  $x, y, z$ , such that  $w = xyz$

and  $|xy| \leq N$ ,

and  $y \neq \epsilon$ ,

and for all  $q \geq 0$ ,  $xy^qz$  is in  $L$ .

(Since must hold for all  $y$ , we **show that it can't hold for any  $y$  that**

**meets**

**the requirements:  $|xy| \leq N$ , and  $y \neq \epsilon$ . To do this:**

**Write out  $w$ :**

**List all the possibilities for  $y$ :**

[1]

[2]

[3]

[4]

For each possibility for  $y$ ,  $xy^qz$  must be in  $L$ , for all  $q$ . So:

**For each possibility for  $y$ , find some value of  $q$  such that  $xy^qz$  is not in  $L$ .** Generally  $q$  will be either 0 or 2.

**y**

**q**

[1]

[2]

[3]

[4]

**Q.E.D.**

# Context-Free Languages and Pushdown Automata

## 1 Context-Free Grammars

Suppose we want to generate a set of strings (a language)  $L$  over an alphabet  $\Sigma$ . How shall we specify our language? One very useful way is to write a grammar for  $L$ . A **grammar** is composed of a set of rules. Each rule may make use of the elements of  $\Sigma$  (which we'll call the **terminal alphabet** or **terminal vocabulary**), as well as an additional alphabet, the **non-terminal alphabet** or **vocabulary**. To distinguish between the terminal alphabet  $\Sigma$  and the non-terminal alphabet, we will use lower-case letters:  $a, b, c$ , etc. for the terminal alphabet and upper-case letters:  $A, B, C, S$ , etc. for the non-terminal alphabet. (But this is just a convention. Any character can be in either alphabet. The only requirement is that the two alphabets be disjoint.)

A grammar generates strings in a language using **rules**, which are instructions, or better, licenses, to replace some non-terminal symbol by some string. Typical rules look like this:

$$S \rightarrow ASa, B \rightarrow aB, A \rightarrow SaSSbB.$$

In context-free grammars, rules have a single non-terminal symbol (upper-case letter) on the left, and any string of terminal and/or non-terminal symbols on the right. So even things like  $A \rightarrow A$  and  $B \rightarrow \epsilon$  are perfectly good context-free grammar rules. What's not allowed is something with more than one symbol to the left of the arrow:  $AB \rightarrow a$ , or a single terminal symbol:  $a \rightarrow Ba$ , or no symbols at all on the left:  $\epsilon \rightarrow Aab$ . The idea is that each rule allows the replacement of the symbol on its left by the string on its right. We call these grammars context free because every rule has just a single nonterminal on its left. We can't add any contextual restrictions (such as  $aAa$ ). So each replacement is done independently of all the others.

To generate strings we start with a designated **start symbol** often  $S$  (for "sentence"), and apply the rules as many times as we please whenever any one is applicable. To get this process going, there will clearly have to be at least one rule in the grammar with the start symbol on the left-hand side. (If there isn't, then the grammar won't generate any strings and will therefore generate  $\emptyset$ , the empty language.) Suppose, however, that the start symbol is  $S$  and the grammar contains both the rules  $S \rightarrow AB$  and  $S \rightarrow aBaa$ . We may apply either one, producing  $AB$  as the "working string" in the first case and  $aBaa$  in the second.

Next we need to look for rules that allow further rewriting of our working string. In the first case (where the working string is  $AB$ ), we want rules with either  $A$  or  $B$  on the left (any non-terminal symbol of the working string may be rewritten by rule at any time); in the latter case, we will need a rule rewriting  $B$ . If, for example, there is a rule  $B \rightarrow aBb$ , then our first working string could be rewritten as  $AaBb$  (the  $A$  stays, of course, awaiting its chance to be replaced), and the second would become  $aABbaa$ .

How long does this process continue? It will necessarily stop when the working string has no symbols that can be replaced. This would happen if either:

- (1) the working string consists entirely of terminal symbols (including, as a special case, when the working string is  $\epsilon$ , the empty string), or
- (2) there are non-terminal symbols in the working string but none appears on the left-hand side of any rule in the grammar (e.g., if the working string were  $AaBb$ , but no rule had  $A$  or  $B$  on the left).

In the first case, but not the second, we say that the working string is **generated by the grammar**. Thus, a grammar generates, in the technical sense, only strings over the terminal alphabet, i.e., strings in  $\Sigma^*$ . In the second case, we have a **blocked** or **non-terminated derivation** but no generated string.

It is also possible that in a particular case neither (1) nor (2) is achieved. Suppose, for example, the grammar contained only the rules  $S \rightarrow Ba$  and  $B \rightarrow bB$ , with  $S$  the start symbol. Then using the symbol  $\Rightarrow$  to connect the steps in the rewriting process, all derivations proceed in the following way:

$$S \Rightarrow Ba \Rightarrow bBa \Rightarrow bbBa \Rightarrow bbbBa \Rightarrow bbbbBa \Rightarrow \dots$$

The working string is always rewriteable (in only one way, as it happens), and so this grammar would not produce any

terminated derivations, let alone any terminated derivations consisting entirely of terminal symbols (i.e., generated strings). Thus this grammar generates the language  $\emptyset$ .

Now let us look at our definition of a context-free grammar in a somewhat more formal way. A context-free grammar (CFG)  $G$  consists of four things:

(1)  $V$ , a finite set (the total alphabet or vocabulary), which contains two subsets,  $\Sigma$  (the *terminal symbols*, i.e., the ones that will occur in strings of the language) and  $V - \Sigma$  (the *nonterminal symbols*, which are just working symbols within the grammar).

(2)  $\Sigma$ , a finite set (the terminal alphabet or terminal vocabulary).

(3)  $R$ , a finite subset of  $(V - \Sigma) \times V^*$ , the set of *rules*. Although each rule is an ordered pair (nonterminal, string), we'll generally use the notion nonterminal  $\rightarrow$  string to describe our rules.

(4)  $S$ , the *start symbol* or *initial symbol*, which can be any member of  $V - \Sigma$ .

For example, suppose  $G = (V, \Sigma, R, S)$ , where

$$V = \{S, A, B, a, b\}, \Sigma = \{a, b\}, \text{ and } R = \{S \rightarrow AB, A \rightarrow aAa, A \rightarrow a, B \rightarrow Bb, B \rightarrow b\}$$

Then  $G$  generates the string  $aaabb$  by the following derivation:

$$(1) \quad S \Rightarrow AB \Rightarrow aAaB \Rightarrow aAaBb \Rightarrow aaaBb \Rightarrow aaabb$$

Formally, given a grammar  $G$ , the two-place relation on strings called "derives in one step" and denoted by  $\Rightarrow$  (or by  $\Rightarrow_G$  if we want to remind ourselves that the relation is relative to  $G$ ) is defined as follows:

$$(u, v) \in \Rightarrow \text{ iff } \exists \text{ strings } w, x, y \in V^* \text{ and symbol } A \in (V - \Sigma) \text{ such that } u = xAy, v = xwy, \text{ and } (A \rightarrow w) \in R.$$

In words, two strings stand in the "derives in one step" relation for a given grammar just in case the second can be produced from the first by rewriting a single non-terminal symbol in a way allowed by the rules of the grammar.

$(u, v) \in \Rightarrow$  is commonly written in infix notation, thus:  $u \Rightarrow v$ .

This bears an obvious relation to the "yields in one step" relation defined on configurations of a finite automaton. Recall that there we defined the "yields in zero or more steps" relation by taking the reflexive transitive closure of the "yields in one step" relation. We'll do that again here, giving us "yields in zero or more steps" denoted by  $\Rightarrow^*$  (or  $\Rightarrow_G^*$ , to be explicit), which holds of two strings iff the second can be derived from the first by finitely many successive applications of rules of the grammar. In the example grammar above:

- $S \Rightarrow AB$ , and therefore also  $S \Rightarrow^* AB$ .
- $S \Rightarrow^* aAaB$ , but not  $S \Rightarrow aAaB$  (since  $aAaB$  cannot be derived from  $S$  in one step).
- $A \Rightarrow aAa$  and  $A \Rightarrow^* aAa$  (This is true even though  $A$  itself is not derivable from  $S$ . If this is not clear, read the definitions of  $\Rightarrow$  and  $\Rightarrow^*$  again carefully.)
- $S \Rightarrow^* S$  (taking zero rule applications), but not  $S \Rightarrow S$  (although the second would be true if the grammar happened to contain the rule  $S \rightarrow S$ , a perfectly legitimate although rather useless rule). Note carefully the difference between  $\rightarrow$ , the connective used in grammar rules, versus  $\Rightarrow$  and  $\Rightarrow^*$ , indicators that one string can be derived from another by means of the rules.

Formally, given a grammar  $G$ , we define a *derivation* to be any sequence of strings

$$w_0 \Rightarrow w_1 \Rightarrow \dots \Rightarrow w_n$$

In other words, a derivation is a finite sequence of strings such that each string, except the first, is derivable in one step from the immediately preceding string by the rules of the grammar. We can also refer to it as a derivation of  $w_n$  from  $w_0$ . Such a derivation is said to be of length  $n$ , or to be a derivation of  $n$  *steps*. (1) above is a 5-step derivation of  $aaabb$  from  $S$  according to the given grammar  $G$ .

Similarly,  $A \Rightarrow aAa$  is a one-step derivation of  $aAa$  from  $A$  by the grammar  $G$ . (Note that derivations do not have to begin with  $S$ , nor indeed do they have to begin with a working string derivable from  $S$ . Thus,  $AA \Rightarrow aAa \Rightarrow aAaa$  is also a well-formed derivation according to  $G$ , and so we are entitled to write  $AA \Rightarrow^* aAaa$ ).

The *strings generated by* a grammar  $G$  are then just those that are (i) derivable from the start symbol, and (ii) composed entirely of terminal symbols. That is,  $G = (V, \Sigma, R, S)$  generates  $w$  iff  $w \in \Sigma^*$  and  $S \Rightarrow^* w$ . Thus, derivation (1) above shows that the string  $aaabb$  is generated by  $G$ . The string  $aAa$ , however, is not generated by  $G$ , even though it is derivable from  $S$ , because it contains a non-terminal symbol. It may be a little harder to see that the string  $bba$  is not generated by  $G$ . One would have to convince oneself that there exists no derivation beginning with  $S$  and ending in  $bba$  according to the rules of  $G$ . (Question: Is this always determinable in general, given any arbitrary context-free grammar  $G$  and string  $w$ ? In other words, can one always tell whether or not a given  $w$  is "grammatical" according to  $G$ ? We'll find out the answer to this later.)

The *language generated by* a grammar  $G$  is exactly the set of all strings generated--no more and no less. The same remarks apply here as in the case of regular languages: a grammar generates a language iff every string in the language is generated by the grammar and no strings outside the language are generated.

And now our final definition (for this section). A language  $L$  is *context free* if and only if there exists a context-free grammar that generates it.

Our example grammar happens to generate the language  $a(aa)^*bb^*$ . To prove this formally would require a somewhat involved argument about the nature of derivations allowed by the rules of  $G$ , and such a proof would not necessarily be easily extended to other grammars. In other words, if you want to prove that a given grammar generates a particular language, you will in general have to make an argument which is rather specific to the rules of the grammar and show that it generates all the strings of the particular language and only those. To prove that a grammar generates a particular string, on the other hand, it suffices to exhibit a derivation from the start symbol terminating in that string. (Question: if such a derivation exists, are we guaranteed that we will be able to find it?) To prove that a grammar does not generate a particular string, we must show that there exists no derivation that begins with the start symbol and terminates in that string. The analogous question arises here: when can we be sure that our search for such a derivation is fruitless and be called off? (We will return to these questions later.)

## 2 Designing Context-Free Grammars

To design a CFG for a language, a helpful heuristic is to imagine generating the strings from the outside in to the middle. The nonterminal that is currently "at work" should be thought of as being at the middle of the string when you are building a string where two parts are interdependent. Eventually the "balancing" regions are done being generated, and the nonterminal that's been doing the work will give way to a different nonterminal (if there's more stuff to be done between the regions just produced) or to some terminal string (often  $\epsilon$ ) otherwise. If parts of a string have nothing to do with each other, do not try to produce them both with one rule. Try to identify the regions of the string that must be generated in parallel due to a correlation between them: they must be generated by the same nonterminal(s). Regions that have no relation between each other can be generated by different nonterminals (and usually should be.)

Here is a series of examples building in complexity. For each one you should generate a few sample strings and build parse trees to get an intuition about what is going on. One notational convention that we'll use to simplify writing language descriptions: If a description makes use of a variable (e.g.,  $a^n$ ), there's an implied statement that the description holds for all integer values  $\geq 0$ .

**Example 1:** The canonical example of a context-free language is  $L = a^n b^n$ , which is generated by the grammar  $G = (\{S, a, b\}, \{a, b\}, R, S)$  where  $R = \{S \rightarrow aSb, S \rightarrow \epsilon\}$ .



Each time an a is generated, a corresponding b is generated. They are created in parallel. The first a, b pair created is the outermost one. The nonterminal S is always between the two regions of a's and b's. Clearly any string  $a^n b^n \in L$  is produced by this grammar, since

$$S \Rightarrow \underbrace{aSb \Rightarrow \dots \Rightarrow a^n S b^n}_{n \text{ steps}} \Rightarrow a^n b^n,$$

Therefore  $L \subseteq L(G)$ .

We must also check that no other strings not in  $a^n b^n$  are produced by the grammar, i.e., we must confirm that  $L(G) \subseteq L$ . Usually this is easy to see intuitively, though you can prove it by induction, typically on the length of a derivation. For illustration, we'll prove  $L(G) \subseteq L$  for this example, though in general you won't need to do this in this class.

*Claim:*  $\forall x, x \in L(G) \Rightarrow x \in L$ . Proof by induction on the length of the derivation of G producing x.

*Base case:* Derivation has length 1. Then the derivation must be  $S \Rightarrow \epsilon$ , and  $\epsilon \in L$ .

*Induction step:* Assume all derivations of length k produce a string in L, and show the claim holds for derivations of length k + 1. A derivation of length k + 1 looks like:

$$S \Rightarrow \underbrace{aSb \Rightarrow \dots \Rightarrow axb}_{k \text{ steps}}$$

for some terminal string x such that  $S \Rightarrow^* x$ . By the induction hypothesis, we know that  $x \in L$  (since x is produced by a derivation of length k), and so  $x = a^n b^n$  for some n (by definition of L). Therefore, the string axb produced by the length k + 1 derivation is  $axb = aa^n b^n b = a^{n+1} b^{n+1} \in L$ . Therefore by induction, we have proved  $L(G) \subseteq L$ .

**Example 2:**  $L = \{xy : |x| = |y| \text{ and } x \in \{a, b\}^* \text{ and } y \in \{c, d\}^*\}$ . (E.g.,  $\epsilon, ac, ad, bc, bd, abaccc \in L$ .) Here again we will want to match a's and b's against c's and d's in parallel. We could use two strategies. In the first,

$$G = (\{S, a, b, c, d\}, \{a, b, c, d\}, R, S) \text{ where } R = \{S \rightarrow aSc, S \rightarrow aSd, S \rightarrow bSc, S \rightarrow bSd, S \rightarrow \epsilon\}.$$

This explicitly enumerates all possible pairings of a, b symbols with c, d symbols. Clearly if the number of symbols allowed in the first and second halves of the strings is n, the number of rules with this method is  $n^2 + 1$ , which would be inefficient for larger alphabets. Another approach is:

$$G = (\{S, L, R, a, b, c, d\}, \{a, b, c, d\}, R, S) \text{ where } R = \{S \rightarrow LSR, S \rightarrow \epsilon, L \rightarrow a, L \rightarrow b, R \rightarrow c, R \rightarrow d\}.$$

(Note that L and R are nonterminals here.) Now the number of rules is  $2n+2$ .

**Example 3:**  $L = \{ww^R : w \in \{a, b\}^*\}$ . Any string in L will have matching pairs of symbols. So it is clear that the CFG  $G = (\{S, a, b\}, \{a, b\}, R, S)$ , where  $R = \{S \rightarrow aSa, S \rightarrow bSb, S \rightarrow \epsilon\}$  generates L, because it produces matching symbols in parallel. How can we prove  $L(G) = L$ ? To do half of this and prove that  $L \subseteq L(G)$  (i.e., every element of L is generated by G), we note that any string  $x \in L$  must either be  $\epsilon$  (which is generated by G (since  $S \Rightarrow \epsilon$ )), or it must be of the form awa or bwb for some  $w \in L$ . This suggests an induction proof on strings:

*Claim:*  $\forall x, x \in L \Rightarrow x \in L(G)$ . Proof by induction on the length of x.

*Base case:*  $\epsilon \in L$  and  $\epsilon \in L(G)$ .

*Induction step:* We must show that if the claim holds for all strings of length k, it holds for all strings of length  $\geq k+2$  (We use k+2 here rather than the more usual k+1 because, in this case, all strings in L have even length. Thus if a string in L has length k, there are no strings in L of length k+1.). If  $|x| = k+2$  and  $x \in L$ , then  $x = awa$  or  $x = bwb$  for some  $w \in L$ .  $|w| = k$ , so, by the induction hypothesis,  $w \in L(G)$ . Therefore  $S \Rightarrow^* w$ . So either  $S \Rightarrow aSa \Rightarrow^* awa$ , and  $x \in L(G)$ , or  $S \Rightarrow bSb \Rightarrow^* bwb$ , and  $x \in L(G)$ .

Conversely, to prove that  $L(G) \subseteq L$ , i.e., that G doesn't generate any bad strings, we would use an induction on the length of a derivation.

*Claim:*  $\forall x, x \in L(G) \Rightarrow x \in L$ . Proof by induction on length of derivation of x.

*Base case:* length 1.  $S \Rightarrow \epsilon$  and  $\epsilon \in L$ .

*Induction step:* Assume the claim is true for derivations of length k, and show the claim holds for derivations of length k+1. A derivation of length k + 1 looks like:

$$S \Rightarrow \underbrace{aS_a \Rightarrow \dots \Rightarrow awa}_{k \text{ steps}}$$

or like

$$S \Rightarrow \underbrace{bS_b \Rightarrow \dots \Rightarrow bw_b}_{k \text{ steps}}$$

for some terminal string  $w$  such that  $S \Rightarrow^* w$ . By the induction hypothesis, we know that  $w \in L$  (since  $w$  is produced by a derivation of length  $k$ ), and so  $x = awa$  is also in  $L$ , by the definition of  $L$ . (Similarly for the second class of derivations that begin with the rule  $S \rightarrow bS_b$ .)

As our example languages get more complex, it becomes harder and harder to write detailed proofs of the correctness of our grammars and we will typically not try to do so.

**Example 4:**  $L = \{a^n b^{2n}\}$ . You should recognize that  $b^{2n} = (bb)^n$ , and so this is just like the first example except that instead of matching  $a$  and  $b$ , we will match  $a$  and  $bb$ . So we want

$$G = (\{S, a, b\}, \{a, b\}, R, S) \text{ where } R = \{S \rightarrow aSbb, S \rightarrow \epsilon\}.$$

If you wanted, you could use an auxiliary nonterminal, e.g.,

$$G = (\{S, B, a, b\}, \{a, b\}, R, S) \text{ where } R = \{S \rightarrow aSB, S \rightarrow \epsilon, B \rightarrow bb\}, \text{ but that is just cluttering things up.}$$

**Example 5:**  $L = \{a^n b^n c^m\}$ . Here, the  $c^m$  portion of any string in  $L$  is completely independent of the  $a^n b^n$  portion, so we should generate the two portions separately and concatenate them together. A solution is

$$G = (\{S, N, C, a, b, c\}, \{a, b, c\}, R, S) \text{ where } R = \{S \rightarrow NC, N \rightarrow aNb, N \rightarrow \epsilon, C \rightarrow cC, C \rightarrow \epsilon\}.$$

This independence buys us freedom: producing the  $c$ 's to the right is completely independent of making the matching  $a^n b^n$ , and so could be done in any manner, e.g., alternate rules like

$$C \rightarrow CC, C \rightarrow c, C \rightarrow \epsilon$$

would also work fine. Thinking modularly and breaking the problem into more manageable subproblems is very helpful for designing CFG's.

**Example 6:**  $L = \{a^n b^m c^n\}$ . Here, the  $b^m$  is independent of the matching  $a^n \dots c^n$ . But it cannot be generated "off to the side." It must be done in the middle, when we are done producing  $a$  and  $c$  pairs. Once we start producing the  $b$ 's, there should be no more  $a, c$  pairs made, so a second nonterminal is needed. Thus we have

$$G = (\{S, B, a, b, c\}, \{a, b, c\}, R, S) \text{ where } R = \{S \rightarrow \epsilon, S \rightarrow aSc, S \rightarrow B, B \rightarrow bB, B \rightarrow \epsilon\}.$$

We need the rule  $S \rightarrow \epsilon$ . We don't need it to end the recursion on  $S$ . We do that with  $S \rightarrow B$ . And we have  $B \rightarrow \epsilon$ . But if  $n = 0$ , then we need  $S \rightarrow \epsilon$  so we don't generate any  $a \dots c$  pairs.

**Example 7:**  $L = a^* b^*$ . The numbers of  $a$ 's and  $b$ 's are independent, so there is no reason to use any rules like  $S \rightarrow aSb$  which create an artificial correspondence. We can independently produce  $a$ 's and  $b$ 's, using

$$G = (\{S, A, B, a, b\}, \{a, b\}, R, S) \text{ where } R = \{S \rightarrow AB, A \rightarrow aA, A \rightarrow \epsilon, B \rightarrow bB, B \rightarrow \epsilon\}$$

But notice that this language is not just context free. It is also regular. So we expect to be able to write a regular grammar (recall the additional restrictions that apply to rules in a regular grammar) for it. Such a grammar will produce  $a$ 's, and then produce  $b$ 's. Thus we could write

$$G = (\{S, B, a, b\}, \{a, b\}, R, S) \text{ where } R = \{S \rightarrow \epsilon, S \rightarrow aS, S \rightarrow bB, B \rightarrow bB, B \rightarrow \epsilon\}.$$

**Example 8:**  $L = \{a^m b^n : m \leq n\}$ . There are several ways to approach this one. One thing we could do is to generate  $a$ 's and  $b$ 's in parallel, and also freely put in extra  $b$ 's. This intuition yields

$$G = (\{S, a, b\}, \{a, b\}, R, S) \text{ where } R = \{S \rightarrow aSb, S \rightarrow Sb, S \rightarrow \epsilon\}.$$

Intuitively, this CFG lets us put in any excess  $b$ 's at any time in the derivation of a string in  $L$ . Notice that to keep the  $S$  between the two regions of  $a$ 's and  $b$ 's, we must use the rule  $S \rightarrow Sb$ ; replacing that rule with  $S \rightarrow bS$  would be incorrect, producing bad strings (allowing extra  $b$ 's to be intermixed with the  $a$ 's).

Another way to approach this problem is to realize that  $\{a^m b^n : m \leq n\} = \{a^m b^{m+k} : k \geq 0\} = \{a^m b^k b^m : k \geq 0\}$ . Therefore, we can produce  $a$ 's and  $b$ 's in parallel, then when we're done, produce some more  $b$ 's. So a solution is

$G = (\{S, B, a, b\}, \{a, b\}, R, S)$  where  $R = \{S \rightarrow \epsilon, S \rightarrow aSb, S \rightarrow B, B \rightarrow bB, B \rightarrow \epsilon\}$ .

Intuitively, this CFG produces the matching a, b pairs, then any extra b's are generated in the middle. Note that this strategy requires two nonterminals since there are two phases in the derivation using this strategy.

Since  $\{a^m b^n : m \leq n\} = \{a^m b^k b^m : k \geq 0\} = \{a^m b^m b^k : k \geq 0\}$ , there is a third strategy: generate the extra b's to the right of the balanced  $a^m b^m$  string. Again the generation of the extra b's is now separated from the generation of the matching portion, so two distinct nonterminals will be needed. In addition, since the two parts are concatenated rather than imbedded, we'll need another nonterminal to produce that concatenation. So we've got

$G = (\{S, M, B, a, b\}, \{a, b\}, R, S)$  where  $R = \{S \rightarrow MB, M \rightarrow aMb, M \rightarrow \epsilon, B \rightarrow bB, B \rightarrow \epsilon\}$ .

**Example 9:**  $L = \{a^{n_1} b^{n_1} \dots a^{n_k} b^{n_k} : k \geq 0\}$ . E.g.,  $\epsilon, abab, aabbaaabbabab \in L$ . Note that  $L = \{a^n b^n\}^*$  which gives a clue how to do this. We know how to produce matching strings  $a^n b^n$ , and we know how to do concatenation of strings. So a solution is

$G = (\{S, M, a, b\}, \{a, b\}, R, S)$  where  $R = \{S \rightarrow MS, S \rightarrow \epsilon, M \rightarrow aMb, M \rightarrow \epsilon\}$ .

Any string  $x = a^{n_1} b^{n_1} \dots a^{n_k} b^{n_k} \in L$  can be generated by the canonical derivation

$$\begin{aligned}
 & S \\
 \Rightarrow^* & /* k \text{ applications of rule } S \rightarrow MS */ \\
 & M^k S \\
 \Rightarrow & /* one application of rule } S \rightarrow \epsilon */ \\
 & M^k \\
 \Rightarrow^* & /* n_1 \text{ applications of rule } M \rightarrow aMb */ \\
 & a^{n_1} M b^{n_1} M^{k-1} \\
 \Rightarrow & /* one application of rule } M \rightarrow \epsilon */ \\
 & a^{n_1} b^{n_1} M^{k-1} \\
 \Rightarrow^* & /* repeating on k-1 remaining M */ \\
 & a^{n_1} b^{n_1} \dots a^{n_k} b^{n_k}
 \end{aligned}$$

Of course the rules could be applied in many different orders.

### 3 Derivations and Parse Trees

Let's again look at the very simple grammar  $G = (V, \Sigma, R, S)$ , where

$V = \{S, A, B, a, b\}, \Sigma = \{a, b\}$ , and  $R = \{S \rightarrow AB, A \rightarrow aAa, A \rightarrow a, B \rightarrow Bb, B \rightarrow b\}$

As we saw in an earlier section,  $G$  can generate the string  $aaabb$  by the following derivation:

$$(1) \quad S \Rightarrow AB \Rightarrow aAaB \Rightarrow aAaBb \Rightarrow aaaBb \Rightarrow aaabb$$

Now let's consider the fact that there are other derivations of the string  $aaabb$  using our example grammar:

$$(2) \quad S \Rightarrow AB \Rightarrow ABb \Rightarrow Abb \Rightarrow aAabb \Rightarrow aaabb$$

$$(3) \quad S \Rightarrow AB \Rightarrow ABb \Rightarrow aAaBb \Rightarrow aAabb \Rightarrow aaabb$$

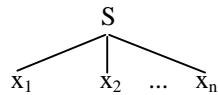
$$(4) \quad S \Rightarrow AB \Rightarrow ABb \Rightarrow aAaBb \Rightarrow aaaBb \Rightarrow aaabb$$

$$(5) \quad S \Rightarrow AB \Rightarrow aAaB \Rightarrow aaaB \Rightarrow aaaBb \Rightarrow aaabb$$

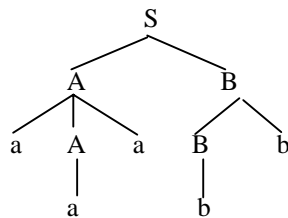
(6)  $S \Rightarrow AB \Rightarrow aAaB \Rightarrow aAaBb \Rightarrow aAabb \Rightarrow aaabb$

If you examine all these derivations carefully, you will see that in each case the same rules have been used to rewrite the same symbols; they differ only in the order in which those rules were applied. For example, in (2) we chose to rewrite the B in  $ABb$  as  $b$  (producing  $Abb$ ) before rewriting the A as  $aAa$ , whereas in (3) the same processes occur in the opposite order. Even though these derivations are technically different (they consist of distinct sequences of strings connected by  $\Rightarrow$ ) it seems that in some sense they should all count as equivalent. This equivalence is expressed by the familiar representations known as *derivation trees* or *parse trees*.

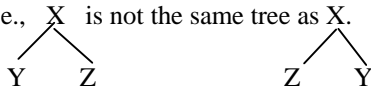
The basic idea is that the start symbol of the grammar becomes the root of the tree. When this symbol is rewritten by a grammar rule  $S \rightarrow x_1x_2\dots x_n$ , we let the tree "grow" downward with branches to each of the new nodes  $x_1, x_2, \dots, x_n$ ; thus:



When one of these  $x_i$  symbols is rewritten, it in turn becomes the "mother" node with branches extending to each of its "daughter" nodes in a similar fashion. Each of the derivations in (1) through (6) would then give rise to the following parse tree:



A note about tree terminology: for us, a tree always has a single root node, and the left-to-right order of nodes is significant; i.e.,  $\begin{matrix} X \\ / \quad \backslash \\ Y \quad Z \end{matrix}$  is not the same tree as  $\begin{matrix} X \\ \backslash \quad / \\ Z \quad Y \end{matrix}$ .

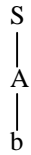


The lines connecting nodes are called *branches*, and their top-to-bottom orientation is also significant. A *mother node* is connected by a single branch to each of the *daughter nodes* beneath it. Nodes with the same mother are called *sisters*, e.g., the topmost A and B in the tree above are sisters, having S as mother.

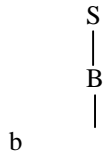
Nodes without daughters are called *leaves*; e.g., each of the nodes labelled with a lower-case letter in the tree above. The string formed by the left-to-right sequence of leaves is called the *yield* ( $aaabb$  in the tree above).

It sometimes happens that a grammar allows the derivations of some string by nonequivalent derivations, i.e., derivations that do not reduce to the same parse tree. Suppose, for example, the grammar contained the rules  $S \rightarrow A$ ,  $S \rightarrow B$ ,  $A \rightarrow b$  and  $B \rightarrow b$ . Then the two following derivations of the string  $b$  correspond to the two distinct parse trees shown below.

$$S \Rightarrow A \Rightarrow b$$



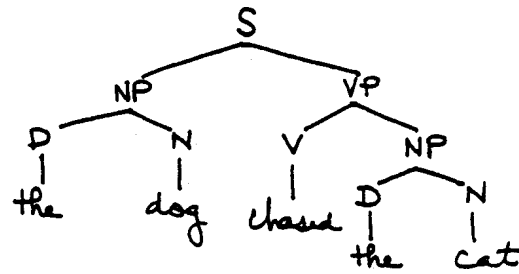
$$S \Rightarrow B \Rightarrow b$$



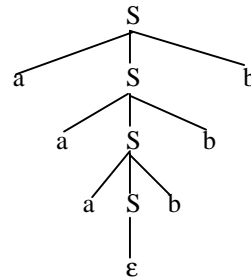
A grammar with this property is said to be *ambiguous*. Such ambiguity is highly undesirable in grammars of programming languages such as C, LISP, and the like, since the parse tree (the syntactic structure) assigned to a string determines its translation into machine language and therefore the sequence of commands to be executed. Designers of programming languages, therefore, take great pains to assure that their grammars (the rules that specify the well-formed strings of the language) are unambiguous. Natural languages, on the other hand, are typically rife with ambiguities (cf. "They are flying planes," "Visiting relatives can be annoying," "We saw her duck," etc.), a fact that makes computer applications such as machine translation, question-answering systems, and so on, maddeningly difficult.

### More Examples of Context-Free Grammars and Parse Trees:

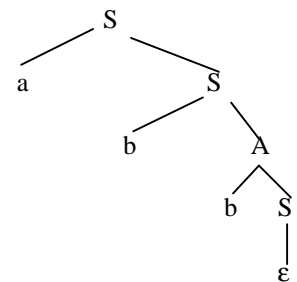
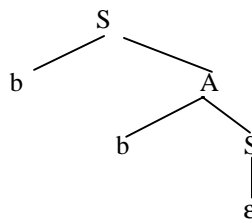
- (1)  $G = (V, \Sigma, R, S)$ , where  
 $V = \{S, NP, VP, D, N, V, \text{chased, the, dog, cat}\}$ ,  
 $\Sigma = \{\text{chased, the, dog, cat}\}$   
 $R = \{S \rightarrow NP VP$   
 $NP \rightarrow D N$   
 $VP \rightarrow V NP$   
 $V \rightarrow \text{chased}$   
 $N \rightarrow \text{dog}$   
 $N \rightarrow \text{cat}$   
 $D \rightarrow \text{the}\}$



- (2)  $G = (V, \Sigma, R, S)$ , where  
 $V = \{S, a, b\}$ ,  $\Sigma = \{a, b\}$ ,  
 $R = \{S \rightarrow aSb$   
 $S \rightarrow \epsilon$   
 $L(G) = \{a^n b^n : n \geq 0\}$



- (3)  $G = (V, \Sigma, R, S)$ ,  
 where  
 $V = \{S, A, a, b\}$ ,  
 $\Sigma = \{a, b\}$ ,  
 $R = \{S \rightarrow aS$   
 $S \rightarrow bA$   
 $A \rightarrow aA$   
 $A \rightarrow bS$   
 $S \rightarrow \epsilon$



- $L(G) = \{w \in \{a, b\}^* :$   
 $w$  contains an even  
 number of b's}

## 4 Designing Pushdown Automata

In a little bit, we will prove that for every grammar  $G$ , there is push down automaton that accepts  $L(G)$ . That proof is constructive. In other words, it describes an algorithm that takes any context-free grammar and constructs the corresponding PDA. Thus, in some sense, we don't need any other techniques for building PDAs (assuming that we already know how to build grammars). But the PDAs that result from this algorithm are often highly nondeterministic. Furthermore, simply using the algorithm gives us no insight into the actual structure of the language we're dealing with. Thus, it is useful to consider how to design PDA's directly; the strategies and insights are different from those we use to design a CFG for a language, and any process that increases our understanding can't be all bad ....

In designing a PDA for a language  $L$ , one of the most useful strategies is to identify the different regions that occur in the strings in  $L$ . As a rule of thumb, each of these regions will correspond to at least one distinct state in the PDA. In this respect, PDAs are very similar to finite state machines. So, for example, just as a finite state machine that accepts  $a^*b^*$  needs two states (one for reading a's and one for reading b's after we're done reading a's), so will a PDA that accepts  $\{a^n b^n\}$  need two states. Recognizing the distinct regions is part of the art, although it is usually obvious.

**Example 1:** In really simple cases, there may not be more than one region. For example, consider  $\{a^n a^n\}$ . What we realize here is that we've really got is just the set of all even length strings of a's, i.e.,  $(aa)^*$ . In this case, the "border" between the first half of the a's and the second half is spurious.

**Example 2:**  $L = \{a^n b^n\}$ . A good thing to try first here is to make a finite state machine for  $a^*b^*$  to get the basic idea of the use of regions. (This principle applies to designing PDA's in general: use the design of a finite state machine to generate the states that are needed just to recognize the basic string structure. This creates the skeleton of your PDA. Then you can add the appropriate stack operations to do the necessary counting.) Clearly you'll need two states since you want to read a's, then read b's. To accept  $L$ , we also need to count the a's in state one in order to match them against the b's in state two. This gives the PDA  $M = (\{s, f\}, \{a, b\}, \{I\}, \Delta, s, \{f\})$ , where  $\Delta =$

```
{ ((s, a, ε), (s, I)),          /* read a's and count them */
  ((s, ε, ε), (f, ε)),         /* guess that we're done with a's and ready to start on b's */
  ((f, b, I), (f, e))}.       /* read b's and compare them to the a's in the stack */
```

(Notice that the stack alphabet need not be in any way similar to the input alphabet. We could equally well have pushed a's, but we don't need to.) This PDA nondeterministically decides when it is done reading a's. Thus one valid computation is

$(a, aabb, \epsilon) \vdash (s, abb, I) \vdash (f, abb, I)$ ,

which is then stuck and so  $M$  rejects along this path. Since a different accepting computation of  $aabb$  exists, this is no problem, but you might want to eliminate the nondeterminism if you are bothered by it. Note that the nondeterminism arises from the  $\epsilon$  transition; we only want to take it if we are done reading a's. The only way to know that there are no more a's is to read the next symbol and see that it's a-b. (This is analogous to unfolding a loop in a program.) One other wrinkle:  $\epsilon \in L$ , so now state  $s$  must be final in order to accept  $\epsilon$ . The resulting deterministic PDA is  $M = (\{s, f\}, \{a, b\}, \{I\}, \Delta, s, \{s, f\})$ , where  $\Delta =$

```
{ ((s, a, ε), (s, I)),          /* read a's and count them */
  ((s, b, I), (f, ε)),         /* only go to second phase if there's a b */
  ((f, b, I), (f, ε))}.       /* read b's and compare them to the a's in the stack */
```

Notice that this DPDA can still get stuck and thus fail, e.g., on input  $b$  or  $aaba$  (i.e., strings that aren't in  $L$ ). Determinism for PDA's simply means that there is at most one applicable transition, not necessarily exactly one.

**Example 3:**  $L = \{a^m b^m c^n d^n\}$ . Here we have two independent concerns, matching the a's and b's, and then matching the c's and d's. Again, start by designing a finite state machine for the language  $L'$  that is just like  $L$  in structure but where we don't care how many of each letter there are. In other words  $a^*b^*c^*d^*$ . It's obvious that this machine needs four states. So our PDA must also have four states. The twist is that we must be careful that there is no unexpected interaction between the two independent parts  $a^m b^m$  and  $c^n d^n$ . Consider the PDA  $M = (\{1,2,3,4\}, \{a,b,c,d\}, \{I\}, \Delta, 1, \{4\})$ , where  $\Delta =$

```
{ ((1, a, ε), (1, I)),          /* read a's and count them */
```

```

((1, ε, ε), (2, ε)), /* guess that we're ready to quit reading a's and start reading b's */
((2, b, I), (2, ε)), /* read b's and compare to a's */
((2, ε, ε), (3, ε)), /* guess that we're ready to quit reading b's and start reading c's */
((3, c, ε), (3, I)) /* read c's and count them */
((3, ε, ε), (4, ε)). /* guess that we're ready to quit reading c's and start reading d's */
((4, d, I), (4, ε)). /* read d's and compare them to c's */

```

It is clear that every string in  $L$  is accepted by this PDA. Unfortunately, some other strings are also, e.g.,  $ad$ . Why is this? Because it's possible to go from state 2 to 3 without clearing off all the  $I$  marks we pushed for the  $a$ 's. That means that the leftover  $I$ 's are available to match  $d$ 's. So this PDA is accepting the language  $\{a^m b^n c^p d^q : m \geq n \text{ and } m + p = n + q\}$ , a superset of  $L$ . E.g., the string  $aabccd$  is accepted.

One way to fix this problem is to ensure that the stack is really cleared before we leave phase 2 and go to phase 3; this must be done using a bottom of stack marker, say  $B$ . This gives  $M = (\{s, 1, 2, 3, 4\}, \{a, b, c, d\}, \{B, I\}, \Delta, s, \{4\})$ , where  $\Delta =$

```

{ ((s, ε, ε), (1, B)), /* push the bottom marker onto the stack */
  ((1, a, ε), (1, I)), /* read a's and count them */
  ((1, ε, ε), (2, ε)), /* guess that we're ready to quit reading a's and start reading b's */
  ((2, b, I), (2, ε)), /* read b's and compare to a's */
  ((2, ε, B), (3, ε)), /* confirm stack is empty, then get ready to start reading c's */
  ((3, c, ε), (3, I)) /* read c's and count them */
  ((3, ε, ε), (4, ε)). /* guess that we're ready to quit reading c's and start reading d's */
  ((4, d, I), (4, ε)). /* read d's and compare them to c's */

```

A different, probably cleaner, fix is to simply use two different symbols for the counting of the  $a$ 's and the  $c$ 's. This gives us  $M = (\{1, 2, 3, 4\}, \{a, b, c, d\}, \{A, C\}, \Delta, 1, \{4\})$ , where  $\Delta =$

```

{ ((1, a, ε), (1, A)), /* read a's and count them */
  ((1, ε, ε), (2, ε)), /* guess that we're ready to quit reading a's and start reading b's */
  ((2, b, A), (2, ε)), /* read b's and compare to a's */
  ((2, ε, ε), (3, ε)), /* guess that we're ready to quit reading b's and start reading c's */
  ((3, c, ε), (3, C)), /* read c's and count them */
  ((3, ε, ε), (4, ε)), /* guess that we're ready to quit reading c's and start reading d's */
  ((4, d, C), (4, ε)). /* read d's and compare them to c's */

```

Now if an input has more  $a$ 's than  $b$ 's, there will be leftover  $A$ 's on the stack and no way for them to be removed later, so that there is no way such a bad string would be accepted.

As an exercise, you want to try making a deterministic PDA for this one.

**Example 4:**  $L = \{a^n b^n\} \cup \{b^n a^n\}$ . Just as with nondeterministic finite state automata, whenever the language we're concerned with can be broken into cases, a reasonable thing to do is build separate PDAs for each of the sublanguages. Then we build the overall machine so that it, each time it sees a string, it nondeterministically guesses which case the string falls into. (For example, compare the current problem to the simpler one of making a finite state machine for the regular language  $a^*b^* \cup b^*a^*$ .) Taking this approach here, we get  $M = (\{s, 1, 2, 3, 4\}, \{a, b\}, \{I\}, \Delta, s, \{2, 4\})$ , where  $\Delta =$

```

{ ((s, ε, ε), (1, ε)), /* guess that this is an instance of a^n b^n */
  ((s, ε, ε), (3, ε)), /* guess that this is an instance of b^n a^n */
  ((1, a, ε), (1, I)), /* a's come first so read and count them */
  ((1, ε, ε), (2, ε)), /* begin the b region following the a's */
  ((2, b, I), (2, ε)), /* read b's and compare them to the a's */
  ((3, b, ε), (3, I)), /* b's come first so read and count them */
  ((3, ε, ε), (4, ε)), /* begin the a region following the b's */
  ((4, a, I), (4, ε)). /* read a's and compare them to the b's */

```

Notice that although  $\epsilon \in L$ , the start state  $s$  is not a final state, but there is a path (in fact two) from  $s$  to a final state.

Now suppose that we want a deterministic machine. We can no longer use this strategy. The  $\epsilon$ -moves must be eliminated by looking ahead. Once we do that, since  $\epsilon \in L$ , the start state must be final. This gives us  $M = (\{s, 1, 2, 3, 4\}, \{a, b\}, \{1\}, \Delta, s, \{s, 2, 4\})$ , where  $\Delta =$

```
{ ((s, a, ε), (1, ε)),      /* if the first character is a, then this is an instance of anbn */
  ((s, b, ε), (3, ε)),      /* if the first character is b, then this is an instance of bnan */
  ((1, a, ε), (1, I)),      /* a's come first so read and count them */
  ((1, b, I), (2, ε)),      /* begin the b region following the a's */
  ((2, b, I), (2, ε)),      /* read b's and compare them to the a's */
  ((3, b, ε), (3, I)),      /* b's come first so read and count them */
  ((3, a, I), (4, ε)),      /* begin the a region following the b's */
  ((4, a, I), (4, ε))}.    /* read a's and compare them to the b's */
```

**Example 5:**  $L = \{ww^R : w \in \{a, b\}^*\}$ . Here we have two phases, the first half and the second half of the string. Within each half, the symbols may be mixed in any particular order. So we expect that a two state PDA should do the trick. See the lecture notes for how it works.

**Example 6:**  $L = \{ww^R : w \in a^*b^*\}$ . Here the two halves of each element of  $L$  are themselves split into two phases, reading a's, and reading b's. So the straightforward approach would be to design a four-state machine to represent these four phases. This gives us  $M = (\{1, 2, 3, 4\}, \{a, b\}, \{a, b\}, \Delta, 1, \{4\})$ , where  $\Delta =$

```
{ ((1, a, ε), (1, a))      /*push a's*/
  ((1, ε, ε), (2, ε)),      /* guess that we're ready to quit reading a's and start reading b's */
  ((2, b, ε), (2, b)),      /* push b's */
  ((2, ε, ε), (3, ε)),      /* guess that we're ready to quit reading the first w and start reading wR */
  ((3, b, b), (3, ε)),      /* compare 2nd b's to 1st b's */
  ((3, ε, ε), (4, ε)),      /* guess that we're ready to quit reading b's and move to the last region of a's */
  ((4, a, a), (4, ε))}.    /* compare 2nd a's to 1st a's */
```

You might want to compare this to the straightforward nondeterministic finite state machine that you might design to accept  $a^*b^*a^*$ .

There are various simplifications that could be made to this machine. First of all, notice that  $L = \{a^mb^nb^na^m\}$ . Next, observe that  $b^nb^n = (bb)^n$ , so that, in effect, the only requirement on the b's is that there be an even number of them. And of course a stack is not even needed to check that. So an alternate solution only needs three states, giving  $M = (\{1, 2, 3\}, \{a, b\}, \{a\}, \{a\}, \Delta, 1, \{3\})$ , where  $\Delta =$

```
{ ((1, a, ε), (1, a))      /*push a's*/
  ((1, ε, ε), (2, ε)),      /* guess that we're ready to quit reading a's and start reading b's */
  ((2, bb, ε), (2, ε)),      /* read bb's */
  ((2, ε, ε), (3, ε)),      /* guess that we're ready to quit reading b's and move on the final group of a's */
  ((3, a, a), (3, ε))}.    /* compare 2nd a's to 1st a's */
```

This change has the fringe benefit of making the PDA more deterministic since there is no need to guess where the middle of the b's occurs. However, it is still nondeterministic.

So let's consider another modification. This time, we go ahead and push the a's and the b's that make up  $w$ . But now we notice that we can match  $w^R$  against  $w$  in a single phase: the required ordering  $b^*a^*$  in  $w^R$  will automatically be enforced if we simply match the input with the stack! So now we have the PDA  $M = (\{1, 2, 3\}, \{a, b\}, \{a, b\}, \Delta, 1, \{3\})$ , where  $\Delta =$

```
{ ((1, a, ε), (1, a))      /*push a's*/
  ((1, ε, ε), (2, ε)),      /* guess that we're ready to quit reading a's and start reading b's */
  ((2, b, ε), (2, b)),      /* push b's */
```



$((2, \epsilon, \epsilon), (3, \epsilon)),$                     /\* guess that we're ready to quit reading the first  $w$  and start reading  $w^R$  \*/  
 $((3, a, a), (3, \epsilon))$                     /\* compare  $w^R$  to  $w^*$  \*/  
 $((3, b, b), (3, \epsilon))\}$ .                    "

Notice that this machine is still nondeterministic. As an exercise, you might try to build a deterministic machine to accept this language. You'll find that it's impossible; you've got to be able to tell when the end of the strings is reached, since it's possible that there aren't any b's in between the a regions. This suggests that there might be a deterministic PDA that accepts  $L$ , and in fact there is. Interestingly, even that is not possible for the less restrictive language  $L = \{ww^R : w \in \{a, b\}^*\}$  (because there's no way to tell without guessing where  $w$  ends and  $w^R$  starts). Putting a strong restriction on string format often makes a language more tractable. Also note that  $\{ww^R : w \in a^*b^+\}$  is accepted by a deterministic PDA; find such a deterministic PDA as an exercise.

**Example 7:** Consider  $L = \{w \in \{a, b\}^* : \#(a, w) = \#(b, w)\}$ . In other words every string in  $L$  has the same number of a's as b's (although the a's and b's can occur in any order). Notice that this language imposes no particular structure on its strings, since the symbols may be mixed in any order. Thus the rule of thumb that we've been using doesn't really apply here. We don't need multiple states for multiple string regions. Instead, we'll find that, other than possible bookkeeping states, one "working" state will be enough.

Sometimes there may be a tradeoff between the degree of nondeterminism in a pda and its simplicity. We can see that in this example. One approach to designing a PDA to solve this problem is to keep a balance on the stack of the excess a's or b's. For example, if there is an a on the stack and we read b, then we cancel them. If, on the other hand, there is an a on the stack and we read another a, we push the new a on the stack. Whenever the stack is empty, we know that we've seen matching number of a's and b's so far. Let's try to design a machine that does this as deterministically as possible. One approach is  $M = (\{s, q, f\}, \{a, b\}, \{a, b, c\}, \Delta, s, \{f\})$ , where  $\Delta =$

1	$((s, \epsilon, \epsilon), (q, c))$	/* Before we do anything else, push a marker, c, on the stack so we'll be able to tell when the stack is empty. Then leave state s so we don't ever do this again.
2	$((q, a, c), (q, ac))$	/* If the stack is empty (we find the bottom c) and we read an a, push c back and then the a (to start counting a's).
3	$((q, a, a), (q, aa))$	/* If the stack already has a's and we read an a, push the new one.
4	$((q, a, b), (q, \epsilon))$	/* If the stack has b's and we read an a, then throw away the top b and the new a.
5	$((q, b, c), (q, bc))$	/* If the stack is empty (we find the bottom c) and we read a b, then start counting b's.
6	$((q, b, b), (q, bb))$	/* If the stack already has b's and we read b, push the new one.
7	$((q, b, a), (q, \epsilon))$	/* If the stack has a's and we read a b, then throw away the top a and the new b.
8	$((q, \epsilon, c), (f, \epsilon))$	/* If the stack is empty then, without reading any input, move to f, the final state.

Clearly we only want to take this transition when we're at the end of the input.

This PDA attempts to solve our problem deterministically, only pushing an a if there is not a b on the stack. In order to tell that there is *not* a b, this PDA has to pop whatever is on the stack and examine it. In order to make sure that there is always something to pop and look at, we start the process by pushing the special marker c onto the stack. (Recall that there is no way to check directly for an empty stack. If we write just  $\epsilon$  for the value of the current top of stack, we'll get a match no what the stack looks like.) Notice, though, that despite our best efforts, we still have a nondeterministic PDA because, at any point in reading an input string, if the number of a's and b's read so far are equal, then the stack consists only of c, and so transition 8  $((q, \epsilon, c), (f, \epsilon))$  may be taken, even if there is remaining input. But if there is still input, then either transition 1 or 5 also applies. The solution to this problem is to add a terminator to  $L$ .

Another thing we could do is to consider a simpler PDA that doesn't even bother trying to be deterministic. Consider  $M = (\{s\}, \{a, b\}, \{a, b\}, \Delta, s, \{s\})$ , where  $\Delta =$

1	$((s, a, \epsilon), (s, a))$	/* If we read an a, push a.
2	$((s, a, b), (s, \epsilon))$	/* Cancel an input a and a stack b.
3	$((s, b, \epsilon), (s, b))$	/* If we read b, push b.
4	$((s, b, a), (s, \epsilon))$	/* Cancel and input b and a stack a.

Now, whenever we're reading a and b is on the stack, there are two applicable transitions: 1, which ignores the b and pushes the a on the stack, and 2, which pops the b and throws away the a (in other words, it cancels the a and b against each other).

Transitions 3 and 4 do the same two things if we're reading  $b$ . It is clear that if we always perform the cancelling transition when we can, we will accept every string in  $L$ . What you might worry about is whether, due to this larger degree of freedom, we might not also be able to wrongly accept some string not in  $L$ . In fact this will not happen because you can prove that  $M$  has the property that, if  $x$  is the string read in so far, and  $y$  is the current stack contents,

$$\#(a, x) - \#(b, x) = \#(a, y) - \#(b, y).$$

This formula is an invariant of  $M$ . We can prove it by induction on the length of the string read so far: It is clearly true initially, before  $M$  reads any input, since  $0 - 0 - 0 - 0$ . And, if it holds before taking a transition, it continues to hold afterward. We can prove this as follows:

Let  $x'$  be the string read so far and let  $y'$  be the contents of the stack at some arbitrary point in the computation. Then let us see what effect each of the four possible transitions has. We first consider:

$((s, a, \epsilon), (s, a))$ : After taking this transition we have that  $x' = xa$  and  $y' = ay$ . Thus we have

$$\begin{aligned} & \#(a, x') - \#(b, x') \\ = & /* x' = xa */ \\ & \#(a, xa) - \#(b, xa) \\ = & /* \#(b, xa) = \#(b, x) */ \\ & \#(a, xa) - \#(b, x) \\ = & \\ & \#(a, x) + 1 - \#(b, x) \\ = & /* induction hypothesis */ \\ & \#(a, y) + 1 - \#(b, y) \\ = & \\ & \#(a, ay) - \#(b, ay) \\ = & /* y' = ay */ \\ & \#(a, y') - \#(b, y') \end{aligned}$$

So the invariant continues to be true for  $x'$  and  $y'$  after the transition is taken. Intuitively, the argument is simply that when this transition is taken, it increments  $\#(a, x)$  and  $\#(a, y)$ , preserving the invariant equation. The three other transitions also preserve the invariant as can be seen similarly:

$((s, a, b), (s, \epsilon))$  increments  $\#(a, x)$  and decrements  $\#(b, y)$ , preserving equality.

$((s, b, \epsilon), (s, b))$  increments  $\#(b, x)$  and  $\#(b, y)$ , preserving equality.

$((s, b, a), (s, \epsilon))$  increments  $\#(b, x)$  and decrements  $\#(a, y)$ , preserving equality.

Therefore, the invariant holds initially, and taking any transitions continues to preserve it, so it is always true, no matter what string is read and no matter what transitions are taken. Why is this a good thing to know? Because suppose a string  $x \notin L$  is read by  $M$ . Since  $x \notin L$ , we know that  $\#(a, x) - \#(b, x) \neq 0$ , and therefore, by the invariant equation, when the whole string  $x$  has been read in, the stack contents  $y$  will satisfy  $\#(a, y) - \#(b, y) \neq 0$ . Thus the stack cannot be empty, and  $x$  cannot be accepted, no matter what sequence of transitions is taken. Thus no bad strings are accepted by  $M$ .

## 5 Context-Free Languages and PDA's

**Theorem:** The context-free languages are exactly the languages accepted by nondeterministic PDA's.

In other words, if you can describe a language with a context-free grammar, you can build a nondeterministic PDA for it, and vice versa. Note here that the class of context-free languages is equivalent to the class of languages accepted by *nondeterministic* PDAs. This is different from what we observed when we were considering regular languages. There we showed that nondeterminism doesn't buy us any power and that we could build a *deterministic* finite state machine for every regular language. Now, as we consider context-free languages, we find that determinism does buy us power: there are languages that are accepted by nondeterministic PDAs for which no deterministic PDA exists. And those languages are context free (i.e., they can be described with context-free grammars). So this theorem differs from the similar theorem that we proved for regular languages and claims equivalence for nondeterministic PDAs rather than deterministic ones.

We'll prove this theorem by construction in two steps: first we'll show that, given a context-free grammar  $G$ , we can construct a PDA for  $L(G)$ . Then we'll show (actually, we'll just sketch this second proof) that we can go the other way and construct, from a PDA that accepts some language  $L$ , a grammar for  $L$ .

**Lemma:** Every context-free language is accepted by some nondeterministic PDA.

To prove this lemma, we give the following construction. Given some CFG  $G = (V, \Sigma, R, S)$ , we construct an equivalent PDA  $M$  in the following way.  $M = (K, \Sigma, \Gamma, \Delta, s, F)$ , where

- $K = \{p, q\}$  (the PDA always has just 2 states)
- $s = p$  ( $p$  is the initial state)
- $F = \{q\}$  ( $q$  is the only final state)
- $\Sigma = \Sigma$  (the input alphabet is the terminal alphabet of  $G$ )
- $\Gamma = V$  (the stack alphabet is the total alphabet of  $G$ )
- $\Delta$  contains
  - (1) the transition  $((p, \epsilon, \epsilon), (q, S))$
  - (2) a transition  $((q, \epsilon, A), (q, \alpha))$  for each rule  $A \rightarrow \alpha$  in  $G$
  - (3) a transition  $((q, a, a), (q, \epsilon))$  for each  $a \in \Sigma$

Notice how closely the machine  $M$  mirrors the structure of the original grammar  $G$ .  $M$  works by using its stack to simulate a derivation by  $G$ . Using the transition created in (1),  $M$  begins by pushing  $S$  onto its stack and moving to its second state,  $q$ , where it will stay for the rest of its operation. Think of the contents of the stack as  $M$ 's expectation for what it must find in order to have seen a legal string in  $L(G)$ . So if it finds  $S$ , it will have found such a string. But if  $S$  could be rewritten as some other sequence  $\alpha$ , then if  $M$  found  $\alpha$  it would also have found a string in  $L(G)$ . All the transitions generated by (2) take care of these options by allowing  $M$  to replace a stack symbol  $A$  by a string  $\alpha$  whenever  $G$  contains the rule  $A \rightarrow \alpha$ . Of course, at some point we actually have to look at the input. That's what  $M$  does in the transitions generated in (3). If the stack contains an expectation of some terminal symbol and if the input string actually contains that symbol,  $M$  consumes the input symbol and pops the expected symbol off the stack (effectively canceling out the expectation with the observed symbol). These steps continue, and if  $M$  succeeds in emptying its stack and reading the entire input string, then the input is accepted.

Let's consider an example. Let  $G = (V = \{S, a, b, c\}, \Sigma = \{a, b, c\}, R = \{S \rightarrow aSa, S \rightarrow bSb, S \rightarrow c\}, S)$ . This grammar generates  $\{xcx^R : x \in \{a, b\}^*\}$ . Carrying out the construction we just described for this example CFG gives the following PDA:

$M = (\{p, q\}, \{a, b, c\}, \{S, a, b, c\}, \Delta, p, \{q\})$ , where

- $\Delta = \{$ 
  - $((p, \epsilon, \epsilon), (q, S))$
  - $((q, \epsilon, S), (q, aSa))$
  - $((q, \epsilon, S), (q, bSb))$
  - $((q, \epsilon, S), (q, c))$
  - $((q, a, a), (q, \epsilon))$
  - $((q, b, b), (q, \epsilon))$
  - $((q, c, c), (q, \epsilon))\}$

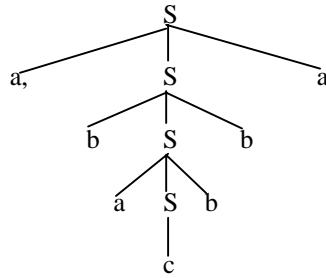
Here is a derivation of the string  $abacaba$  by  $G$ :

(1)  $S \Rightarrow aSa \Rightarrow abSba \Rightarrow abaSaba \Rightarrow abacaba$

And here is a computation by  $M$  accepting that same string:

(2)  $(p, abacaba, \epsilon) \vdash (q, abacaba, S) \vdash (q, abacaba, aSa) \vdash (q, bacaba, Sa) \vdash (q, bacaba, bSba) \vdash (q, acaba, Sba) \vdash (q, acaba, aSaba) \vdash (q, caba, Saba) \vdash (q, caba, caba) \vdash (q, aba, aba) \vdash (q, ba, ba) \vdash (q, a, a) \vdash (q, \epsilon, \epsilon)$

If you look at the successive stack contents in computation (2) above, you will see that they are, in effect, tracing out a derivation tree for the string abacaba:



M is alternately extending the tree and checking to see if leaves of the tree match the input string. M is thus acting as a *top-down parser*. A parser is something that determines whether a presented string is generated by a given grammar (i.e., whether the string is *grammatical* or *well-formed*), and, if it is, calculates a syntactic structure (in this case, a parse tree) assigned to that string by the grammar. Of course, the machine M that we have just described does not in fact produce a parse tree, although it could be made to do so by adding some suitable output devices. M is thus not a parser but a *recognizer*. We'll have more to say about parsers later, but we can note here that parsers play an important role in many kinds of computer applications including compilers for programming languages (where we need to know the structure of each command), query interpreters for database systems (where we need to know the structure of each user query), and so forth.

Note that M is properly non-deterministic. From the second configuration in (2), we could have gone to (q, abacaba, bSb) or to (q, abacaba, c), for example, but if we'd done either of those things, M would have reached a dead end. M in effect has to guess which one of a group of applicable rules of G, if any, is the right one to derive the given string. Such guessing is highly undesirable in the case of most practical applications, such as compilers, because their operation can be slowed down to the point of uselessness. Therefore, programming languages and query languages (which are almost always context-free, or nearly so) are designed so that they can be parsed deterministically and therefore compiled or interpreted in the shortest possible time. A lot of attention has been given to this problem in Computer Science, as you will learn if you take a course in compilers. On the other hand, natural languages, such as English, Japanese, etc., were not "designed" for this kind of parsing efficiency. So, if we want to deal with them by computer, as for example, in machine translation or information retrieval systems, we have to abandon any hope of deterministic parsing and strive for maximum non-deterministic efficiency. A lot of effort has been devoted to these problems as well, as you will learn if you take a course in computational linguistics.

To complete the proof of our lemma, we need to prove that  $L(M) = L(G)$ . The proof is by induction and is reasonably straightforward. We'll omit it here, and turn instead to the other half of the theorem:

**Lemma:** If M is a non-deterministic PDA, there is a context-free grammar G such that  $L(G) = L(M)$ .

Again, the proof is by construction. Unfortunately, this time the construction is anything but natural. We'd never want actually to do it. We just care that the construction exists because it allows us to prove this crucial result. The basic idea behind the construction is to build a grammar that has the property that if we use it to create a leftmost derivation of some string s then we will have simulated the behavior of M while reading s. The nonterminals of the grammar are things like  $\langle s, Z, f \rangle$  (recall that we can use any names we want for our nonterminals). The reason we use such strange looking nonterminals is to make it clear what each one corresponds to. For example,  $\langle s, Z, f \rangle$  will generate all strings that M could consume in the process of moving from state s with Z on the stack to state f having popped Z off the stack.

To construct G from M, we proceed in two steps: First we take our original machine M and construct a new "simple" machine M' (see below). We do this so that there will be fewer cases to consider when we actually do the construction of a grammar from a machine. Then we build a grammar from M'.

A pda M is *simple* iff:

- (1) There are no transitions into the start state, and
- (2) Whenever  $((q, a, \beta), (p, \gamma))$  is a transition in M and q is not the start state, then  $\beta \in \Gamma$  and  $|\gamma| \leq 2$ .

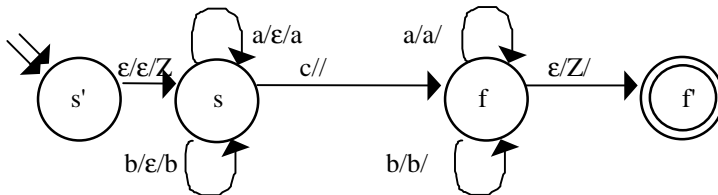
In other words,  $M$  is simple if it always consults its topmost stack symbol (and no others) and replaces that symbol either with 0, 1, or 2 new symbols. We need to treat the start state separately since of course when  $M$  starts, its stack is empty and there is nothing to consult. But we do need to guarantee that the start state can't bypass the restriction of (2) if it also functions as something other than the start state i.e., it is part of a loop. Thus constraint (1).

Although not all machines are simple, there is an algorithm to construct an equivalent simple machine from any machine  $M$ . Thus the fact that our grammar construction algorithm will work only on simple machines in no way limits the applicability of the lemma that says that for any machine there is an *equivalent* grammar.

Given any PDA  $M$ , we construct an equivalent simple PDA  $M'$  as follows:

(1) Let  $M' = M$ .

(2) Add to  $M'$  a new start state  $s'$  and a new final state  $f'$ . Add a transition from  $s'$  to  $M$ 's original start state that consumes no input and pushes a special "stack bottom" symbol  $Z$  onto the stack. Add transitions from all of  $M$ 's original final states to  $f'$ . These transitions should consume no input but they should pop the bottom of stack symbol  $Z$  from the stack. For example, if we start with a straightforward two-state PDA that accepts  $wc^R$ , then this step produces:



(3) (a) Assure that  $|\beta| \leq 1$ . In other words, make sure that no transition looks at more than one symbol on the stack. It is easy to do this. If there are any transitions in  $M'$  that look at two or more symbols, break them down into multiple transitions that examine one symbol apiece.

(b) Assure that  $|\gamma| \leq 1$ . In other words, make sure that each transition pushes no more than one symbol onto the stack. (The rule for simple allows us to push 2, but you'll see why we restrict to 1 at this point in a minute.) Again, if  $M'$  has any transitions that push more than one symbol, break them apart into multiple steps.

(c) Assure that  $|\beta| = 1$ . We already know that  $|\beta|$  isn't greater than 1. But it could be zero. If there are any transitions that don't examine the stack at all, then change them so that they pop off the top symbol, ignore it, and push it right back on. When we do this, we will increase by one the length of the string that gets pushed onto the stack. Now you can see why we did step (b) as we did. If, after completing (b) we never pushed more than one symbol, we can go ahead and do (c) and still be assured that we never push more than two symbols (which is what we require for  $M'$  to be simple).

We'll omit the proof that this procedure does in fact produce a new machine  $M'$  that is simple and equivalent to  $M$ .

Once we have a simple machine  $M'$  ( $K', \Sigma', \Gamma', \Delta', s', f'$ ) derived from our original machine  $M$  ( $K, \Sigma, \Gamma, \Delta, s, F$ ), we are ready to construct a grammar  $G$  for  $L(M')$  (and thus, equivalently, for  $L(M)$ ). We let  $G = (V, \Sigma, R, S)$ , where  $V$  contains a start symbol  $S$ , all the elements of  $\Sigma$ , and a new nonterminal symbol  $\langle q, A, p \rangle$  for every  $q$  and  $p$  in  $K'$  and every  $A = \epsilon$  or any symbol in the stack alphabet of  $M'$  (which is the stack alphabet of  $M$  plus the special bottom of stack marker). The tricky part is the construction of  $R$ , the rules of  $G$ .  $R$  contains all the following rules (although in fact most will be useless in the sense that the nonterminal symbol on the left hand side will never be generated in any derivation that starts with  $S$ ):

(1) The special rule  $S \rightarrow \langle s, Z, f' \rangle$ , where  $s$  is the start state of the original machine  $M$ ,  $Z$  is the special "bottom of stack" symbol that  $M'$  pushes when it moves from  $s'$  to  $s$ , and  $f'$  is the new final state of  $M'$ . This rule says that to be a string in  $L(M)$  you must be a string that  $M'$  can consume if it is started in state  $s$  with  $Z$  on the top of the stack and it makes it to state  $f'$  having popped  $Z$  off the stack. All the rest of the rules will correspond to the various paths by which  $M'$  might do that.

(2) Consider each transition  $((q, a, B), (r, C))$  of  $M'$  where  $a$  is either  $\epsilon$  or a single input symbol and  $C$  is either a single symbol or  $\epsilon$ . In other words, each transition of  $M'$  that pushes zero or one symbol onto the stack. For each such transition and each state  $p$  of  $M'$ , we add the rule

$$\langle q, B, p \rangle \rightarrow a \langle r, C, p \rangle.$$

Read these rule as saying that one way in which  $M'$  can go from  $q$  to  $p$  and pop  $B$  off the stack is by consuming an  $a$ , going to

state  $r$ , pushing a  $C$  on the stack (all of which are specified by the transition we're dealing with), then getting eventually to  $p$  and popping off the stack the  $C$  that the transition specifies must be pushed. Think of these rules this way. The transition that motivates them tells us how to make a single move from  $q$  to  $r$  while consuming the input symbol  $a$  and popping the stack symbol  $B$ . So think about the strings that could drive  $M'$  from  $q$  to some arbitrary state  $p$  (via this transition) and pop  $B$  from the stack in the process. They include all the strings that start with  $a$  and are followed by the strings that can drive  $M'$  from  $r$  on to  $p$  provided that they also cause the  $C$  that got pushed to be dealt with and popped. Note that of course we must also pop anything else we push along the way, but we don't have to say that explicitly since if we haven't done that we can't get to  $C$  to pop it.

(3) Next consider each transition  $((q, a, B), (r, CD))$  of  $M'$ , where  $C$  and  $D$  are stack symbols. In other words, consider every transition that pushes two symbols onto the stack. (Recall that since  $M'$  is simple, we only have to consider the cases of 0, 1, or 2 symbols being pushed.) Now consider all pairs of states  $v$  and  $w$  in  $K'$  (where  $v$  and  $w$  are not necessarily distinct). For all such transitions and pairs of states, construct the rule

$$\langle q, B, v \rangle \rightarrow a \langle r, C, w \rangle \langle w, D, v \rangle$$

These rules are a bit more complicated than the ones that were generated in (2) just because they describe computations that involve two intermediate states rather than one, but they work the same way.

(4) For every state  $q$  in  $M'$ , we add the rule

$$\langle q, \epsilon, q \rangle \rightarrow \epsilon$$

These rules let us get rid of spurious nonterminals so that we can actually produce strings composed solely of terminal symbols. They correspond to the fact that  $M'$  can (trivially) get from a state back to itself while popping nothing simply by doing nothing (i.e., reading the empty string).

See the lecture notes for an example of this process in action. As you'll notice, the grammars that this procedure generates are very complicated, even for very simple machines. From larger machines, one would get truly enormous grammars (most of whose rules turn out to be useless, as a matter of fact). So, if one is presented with a PDA, the best bet for finding an equivalent CFG is to figure out the language accepted by the PDA and then proceed intuitively to construct a CFG that generates that language.

We'll omit here the proof that this process does indeed produce a grammar  $G$  such that  $L(G) = L(M)$ .

## 6 Parsing

Almost always, the reason we care about context-free languages is that we want to build programs that "interpret" or "understand" them. For example, programming languages are context free. So are most data base query languages. Command languages that need capabilities (such as matching delimiters) that can't exist in simpler, regular languages are also context free.

The interpretation process for context free languages generally involves three parts (although these logical parts may be interleaved in various ways in the interpretation program):

1. Lexical analysis, in which individual characters are combined, generally using finite state machine techniques, to form the building blocks of the language.
2. Parsing, in which a tree structure is assigned to the string.
3. Semantic interpretation, in which "meaning", often in the form of executable code, is attached to the nodes of the tree and thus to the entire string itself.

For example, consider the input string "orders := orders + 1;", which might be a legal string in any of a number of programming languages. Lexical analysis first divides this string of characters into a sequence of six *tokens*, each of which corresponds to a basic unit of meaning in the language. The tokens generally contain two parts, an indication of what kind of thing they are and the actual value of the string that they matched. The six tokens are (with the kind of token shown, followed by its value in parentheses):

$$\langle \text{id} \rangle (\text{orders}) \quad := \quad \langle \text{id} \rangle (\text{orders}) \quad \langle \text{op} \rangle (+) \quad \langle \text{id} \rangle (1) \quad ;$$

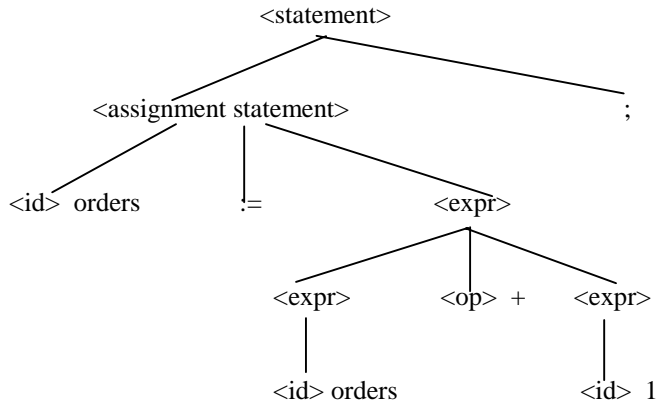
Assume that we have a grammar for our language that includes the following rules:

```

<statement> → <assignment statement> ;
<statement> → <loop statement> ;
<assignment statement> → <id> := <expr>
<expr> → <expr> <op> <expr>
<expr> → <id>

```

Using this grammar and the string of tokens produced above, parsing assigns to the string a tree structure like



Finally, we need to assign a meaning to this string. If we attach appropriate code to each node of this tree, then we can execute this statement by doing a postorder traversal of the tree. We start at the top node, <statement> and traverse its left branch, which takes us to <assignment statement>. We go down its left branch, and, in this case, we find the address of the variable orders. We come back up to <assignment statement>, and then go down its middle branch, which doesn't tell us anything that we didn't already know from the fact that we're in an assignment statement. But we still need to go down the right branch to compute the value that is to be stored. To do that, we start at <expr>. To get its value, we must examine its subtrees. So we traverse its left branch to get the current value for orders. We then traverse the middle branch to find out what operation to perform, and then the right branch and get 1. We hand those three things back up to <expr>, which applies the + operator and computes a new value, which we then pass back up to <assignment statement> and then to <statement>.

Lexical analysis is a straightforward process that is generally done using a finite state machine. Semantic interpretation can be arbitrarily complex, depending on the language, as well as other factors, such as the degree of optimization that is desired. Parsing, though, is in the middle. It's not completely straightforward, particularly if we are concerned with efficiency. But it doesn't need to be completely tailored to the individual application. There are some general techniques that can be applied to a wide variety of context-free languages. It is those techniques that we will discuss briefly here.

## 6.1 Parsing as Search

Recall that a parse tree for a string in a context-free language describes the set of grammar rules that were applied in the derivation of the string (and thus the syntactic structure of the string). So to parse a string we have to find that set of rules. How shall we do it? There are two main approaches:

1. Top down, in which we start with the start symbol of the grammar and work forward, applying grammar rules and keeping track of what we're doing, until we succeed in deriving the string we're interested in.
2. Bottom up, in which we start with the string we're interested in. In this approach, we apply grammar rules "backwards". So we look for a rule whose right hand side matches a piece of our string. We "apply" it and build a small subtree that will eventually be at the bottom of the parse tree. For example, given the assignment statement we looked at above, we might start by building the tree whose root is <expr> and whose (only) leaf is <id> orders. That gives us a new "string" to work with, which in this case would be orders := <expr> <op> <id>(1). Now we look for a grammar rule that matches part of this "string" and apply it. We continue until we apply a rule whose left hand side is the start symbol. At that point, we've got a complete tree.

Whichever of these approaches we choose, we'd like to be as efficient as possible. Unfortunately, in many cases, we're

forced to conduct a search, since at any given point it may not be possible to decide which rule to apply. There are two reasons why this might happen:

- Our grammar may be ambiguous and there may actually be more than one legal parse tree for our string. We will generally try to design languages, and grammars for them, so that this doesn't happen. If a string has more than one parse tree, it is likely to have more than one meaning, and we rarely want to use languages where users can't predict the meaning of what they write.
- There may be only a single parse tree but it may not be possible to know, without trying various alternatives and seeing which ones work, what that tree should be. This is the problem we'll try to solve with the introduction of various specific parsing techniques.

## 6.2 Top Down Parsing

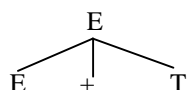
To get a better feeling for why a straightforward parsing algorithm may require search, let's consider again the following grammar for arithmetic expressions:

- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow \text{id}$

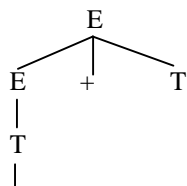
Let's try to do a top down parse, using this grammar, of the string `id + id * id`. We will begin with a tree whose only node is  $E$ , the start symbol of the grammar. At each step, we will attempt to expand the leftmost leaf nonterminal in the tree. Whenever we rewrite a nonterminal as a terminal (for example, when we rewrite  $F$  as `id`), we'll climb back up the tree and down another branch, each time expanding the leftmost leaf nonterminal). We could do it some other way. For example, we could always expand the rightmost nonterminal. But since we generally read the input string left to right, it makes sense to process the parse tree left to right also.

No sooner do we get started on our example parse but we're faced with a choice. Should we expand  $E$  by applying rule (1) or rule (2)? If we choose rule (1), what we're doing is choosing the interpretation in which `+` is done after `*` (since `+` will be at the top of the tree). If we choose rule (2), we're choosing the interpretation in which `*` is done after `+` (since `*` will be nearest the top of the tree, which we'll detect at the next step when we have to find a way to rewrite  $T$ ). We know (because we've done this before and because we know that we carefully crafted this grammar to force `*` to have higher precedence than `+`) that if we choose rule (2), we'll hit a dead end and have to back up, since there will be no way to deal with `+` inside  $T$ .

Let's just assume for the moment that our parser also knows the right thing to do. It then produces



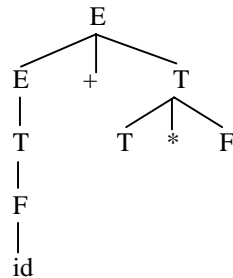
Since  $E$  is again the leftmost leaf nonterminal, we must again choose how to expand it. This time, the right thing to do is to choose rule (2), which will rewrite  $E$  as  $T$ . After that, the next thing to do is to decide how to rewrite  $T$ . The right thing to do is to choose rule (4) and rewrite  $T$  as  $F$ . Then the next thing to do is to apply rule (6) and rewrite  $F$  as `id`. At this point, we've generated a terminal symbol. So we read an input symbol and compare it to the one we've generated. In this case, it matches, so we can continue. If it didn't match, we'd know we'd hit a dead end and we'd have to back up and try another way of expanding one of the nodes higher up in the tree. But since we found a match, we can continue. At this point, the tree looks like







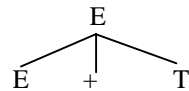
Since we matched a terminal symbol (id), the next thing to do is to back up until we find a branch that we haven't yet explored. We back all the way up to the top E, then down its center branch to +. Since this is a terminal symbol, we read the next input symbol and check for a match. We've got one, so we continue by backing up again to E and taking the third branch, down to T. Now we face another choice. Should we apply rule (3) or rule (4). Again, being smart, we'll choose to apply rule (3), producing



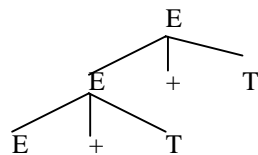
The rest of the parse is now easy. We'll expand T to F and then match the second id. Then we'll match F to the last id.

But how can we make our parser know what we knew?

In this case, one simple heuristic we might try is to consider the rules in the order in which they appear in the grammar. That will work for this example. But suppose the input had been `id * id * id`. Now we need to choose rule (2) initially. And we're now in big trouble if we always try rule (1) first. Why? Because we'll never realize we're on the wrong path and back up and try rule (2). If we choose rule (1), then we will produce the partial parse tree



But now we again have an E to deal with. If we choose rule (1) again, we have



And then we have another E, and so forth. The problem is that rule (1) contains *left recursion*. In other words, a symbol, in this case E, is rewritten as a sequence whose first symbol is identical to the symbol that is being rewritten.

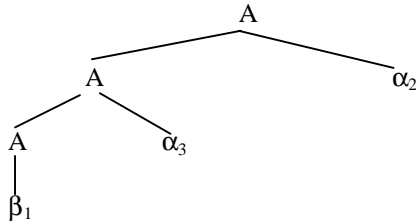
We can solve this problem by rewriting our grammar to get rid of left recursion. There's an algorithm to do this that always works. We do the following for each nonterminal A that has any left recursive rules. We look at all the rules that have A on their left hand side, and we divide them into two groups, the left recursive ones and the other ones. Then we replace each rule with another related rule as shown in the following table:

	<b>Original rules</b>	<b>New rules</b>
<b>Left recursive rules:</b>	$A \rightarrow A\alpha_1$	$A' \rightarrow \alpha_1 A'$
	$A \rightarrow A\alpha_2$	$A' \rightarrow \alpha_2 A'$
	$A \rightarrow A\alpha_3 \quad \dots$	$A' \rightarrow \alpha_3 A' \quad \dots$
	$A \rightarrow A\alpha_n$	$A' \rightarrow \alpha_n A'$
		$A' \rightarrow \epsilon$

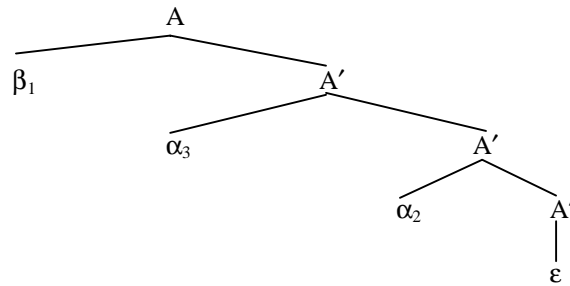
**Non-left recursive rules:**

$A \rightarrow \beta_1$		$A \rightarrow \beta_1 A'$	
$A \rightarrow \beta_2$	...	$A \rightarrow \beta_2 A'$	...
$A \rightarrow \beta_n$		$A \rightarrow \beta_n A'$	

The basic idea is that, using the original grammar, in any successful parse,  $A$  may be expanded some arbitrary number of times using the left recursive rules, but if we're going to get rid of  $A$  (which we must do to derive a terminal string), then we must eventually apply one of the nonrecursive rules. So, using the original grammar, we might have something like



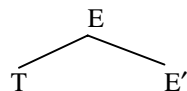
Notice that, whatever  $\beta_1$ ,  $\alpha_3$ , and  $\alpha_2$  are,  $\beta_1$ , which came from one of the nonrecursive rules, comes first. Now look at the new set of rules in the right hand column above. They say that  $A$  must be rewritten as a string that starts with the right hand side of one of the nonrecursive rules (i.e., some  $\beta_i$ ). But, if any of the recursive rules had been applied first, then there would be further substrings, after the  $\beta_i$ , derived from those recursive rules. We introduce the new nonterminal  $A'$  to describe what those things could look like, and we write rules, based on the original recursive rules, that tell us how to rewrite  $A'$ . Using this new grammar, we'd get a parse tree for  $\beta_1 \alpha_3 \alpha_2$  that would look like



If we apply this transformation algorithm to our grammar for arithmetic expressions, we get

- (1)  $E' \rightarrow + T E'$
- (1')  $E' \rightarrow \epsilon$
- (2)  $E \rightarrow T E'$
- (3)  $T' \rightarrow * F T'$
- (3')  $T' \rightarrow \epsilon$
- (4)  $T \rightarrow F T'$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow \text{id}$

Now let's return to the problem of parsing  $\text{id} + \text{id} * \text{id}$ . This time, there is only a single way to expand the start symbol,  $E$ , so we produce, using rule (2),



Now we need to expand  $T$ , and again, there is only a single choice. If you continue with this example, you'll see that if you have the ability to peek one character ahead in the input (we'll call this character the *lookahead character*), then it's possible to know, at each step in the parsing process, which rule to apply.

You'll notice that this parse tree assigns a quite different structure to the original string. This could be a serious problem when we get ready to assign meaning to the string. In particular, if we get rid of left recursion in our grammar for arithmetic expressions, we'll get parse trees that associate right instead of left. For example, we'll interpret  $a + b + c$  as

$(a + b) + c$  using the original grammar, but

$a + (b + c)$  using the new grammar.

For this and various other reasons, it often makes sense to change our approach and parse bottom up, rather than top down.

### 6.3 Bottom Up Parsing

Now let's go back to our original grammar for arithmetic expressions:

- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow \text{id}$

Let's try again to parse the string  $\text{id} + \text{id} * \text{id}$ , this time working bottom up. We'll scan the input string from left to right, just as we've always done with all the automata we've built. A bottom up parser can perform two basic operations:

1. Shift an input symbol onto the parser's stack.
2. Reduce a string of symbols from the top of the stack to a nonterminal symbol, using one of the rules of the grammar. Each time we do this, we also build the corresponding piece of the parse tree.

When we start, the stack is empty, so our only choice is to get the first input symbol and shift it onto the stack. The first input symbol is  $\text{id}$ , so it goes onto the stack. Next, we have a choice. We can either use rule (6) to reduce  $\text{id}$  to  $F$ , or we can get the next input symbol and push it onto the stack. It's clear that we need to apply rule (6) now. Why? There are no other rules that can consume an  $\text{id}$  directly. So we have to do this reduction before we can do anything else with  $\text{id}$ . But could we wait and do it later? No, because reduction always applies to the symbols at the top of the stack. If we push anything on before we reduce  $\text{id}$ , we'll never again get  $\text{id}$  at the top of the stack. So it will just sit there, unable to participate in any rules. So the next thing we need to do is to reduce  $\text{id}$  to  $F$ , giving us a stack containing just  $F$ , and the parse tree (remember we're building up from the bottom):

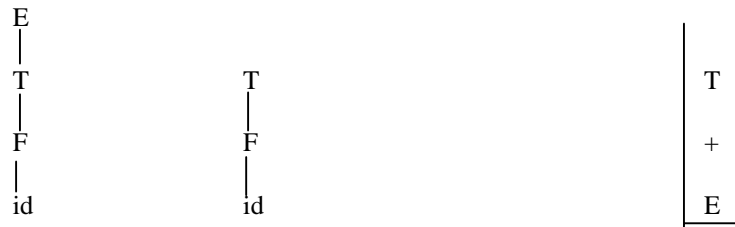
$$\begin{array}{c} F \\ | \\ \text{id} \end{array}$$

Before we continue, let's observe that the reasoning we just did is going to be the basis for the design of a "smart" deterministic bottom up parser. Without that reasoning, a dumb, brute force parser would have to consider both paths at this first choice point: the one we took, as well as the one that fails to reduce and instead pushes  $+$  onto the stack. That second path will dead end eventually, so even a brute force parser will eventually get the right answer. But for efficiency, we'd like to build a deterministic parser if we can. We'll return to the question of how we do that after we finish with this example so we can see all the places we're going to have to make our parser "smart".

At this point, the parser's stack contains  $F$  and the remaining input is  $+ \text{id} * \text{id}$ . Again we must choose between reducing the top of the stack or pushing on the next input symbol. Again, by looking ahead and analyzing the grammar, we can see that eventually we will need to apply rule (1). To do so, the first  $\text{id}$  will have to have been promoted to a  $T$  and then to an  $E$ . So let's next reduce by rule (4) and then again by rule (2), giving the parse tree and stack:



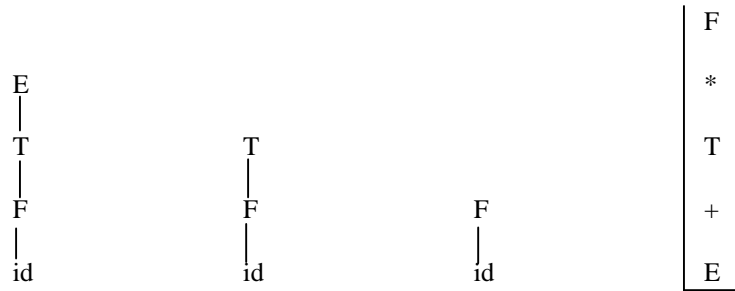
At this point, there are no further reductions to consider, since there are no rules whose right hand side is just E. So we must consume the next input symbol + and push it onto the stack. Now, again, there are no available reductions. So we read the next symbol, and the stack then contains id + E (we'll write the stack so that we push onto the left). Again, we need to promote id before we can do anything else, so we promote it to F and then to T. Now we've got:



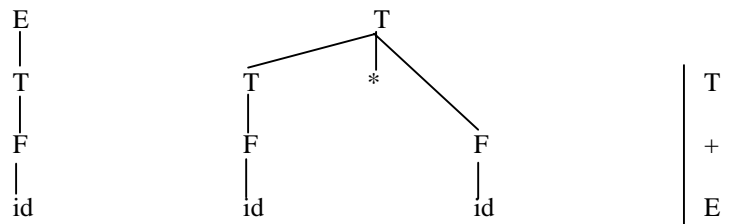
Notice that we've now got two parse tree fragments. Since we're working up from the bottom, we don't know yet how they'll get put together. The next thing we have to do is to choose between reducing the top three symbols on the stack (T + E) to E using rule (1) or shifting on the next input symbol. By the way, don't be confused about the order of the symbols here. We'll always be matching the right hand sides of the rules reversed because the last symbol we read (and thus the right most one we'll match) is at the top of the stack.

Okay, so what should we choose to do, reduce or shift? This is the first choice we've had to make where there isn't one correct answer for all input strings. When there was just one universally correct answer, we could compute it simply by examining the grammar. Now we can't do that. In the example we're working with, we don't want to do the reduction, since the next operator is \*. We want the parse tree to correspond to the interpretation in which \* is applied before +. That means that + must be at the top of the tree. If we reduce now, it will be at the bottom. So we need to shift \* on and do a reduction that will build the multiplication piece of the parse tree before we do a reduction involving +. But if the input string had been id + id + id, we'd want to reduce now in order to cause the first + to be done first, thus producing left associativity. So we appear to have reached a point where we'll have to branch. Since our grammar won't let us create the interpretation in which we do the + first, if we choose that path first, we'll eventually hit a dead end and have to back up. We'd like not to waste time exploring dead end paths, however. We'll come back to how we can make a parser smart enough to do that later. For now, let's just forge ahead and do the right thing.

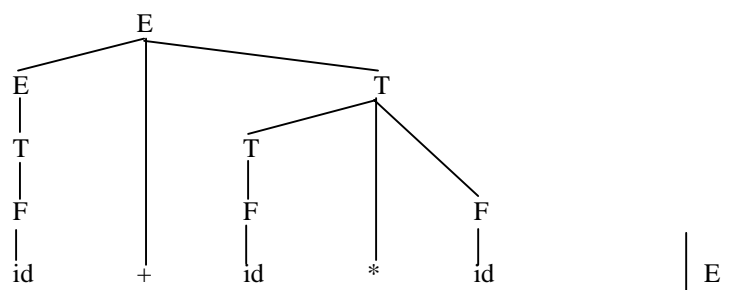
As we said, what we want to do here is not to reduce but instead to shift \* onto the stack. So the stack now contains \* T + E. At this point, there are no available reductions (since there are no rules whose right hand side contains \* as the last symbol), so we shift the next symbol, resulting in the stack id \* T + E. Clearly we have to promote id to F (following the same argument that we used above), so we've got



Next, we need to reduce (since there aren't any more input symbols to shift), but now we have another decision to make: should we reduce the top F to T, or should we reduce the top three symbols, using rule (3) to T? The right answer is to use rule (3), producing:



Finally, we need to apply rule (1), to produce the single symbol E on the top of the stack, and the parse tree:



In a bottom up parse, we're done when we consume all the input and produce a stack that contains a single symbol, the start symbol. So we're done (although see the class notes for an extension of this technique in which we add to the input and end-of-input symbol \$ and consume it as well).

Now let's return to the question of how we can build a parser that makes the right choices at each step of the parsing process. As we did the example parse above, there were two kinds of decisions that we had to make:

- Whether to shift or reduce (we'll call these shift/reduce conflicts), and
- Which of several available reductions to perform (we'll call these reduce/reduce conflicts).

Let's focus first on shift/reduce conflicts. At least in this example, it was always possible to make the right decision on these conflicts if we had two kinds of information:

- A good understanding of what is going on in the grammar. For example, we noted that there's nothing to be done with a raw id that hasn't been promoted to an F.
- A peek at the next input symbol (the one that we're considering shifting), which we call the lookahead symbol. For example, when we were trying to decide whether to reduce  $T + E$  or shift on the next symbol, we looked ahead and saw that the next symbol was \*. Since we know that \* has higher precedence than +, we knew not to reduce +, but rather to wait and deal with \* first.

So we as people can be smart and do the right thing. The important question is, "Can we build a parser that is smart and does the right thing?" The answer is yes. For simple grammars, like the one we're using, it's fairly straightforward to do so. For more complex grammars, the algorithms that are needed to produce a correct deterministic parser are way beyond the scope of this class. In fact, they're not something most people ever want to deal with. And that's okay because there are powerful tools for building parsers. The input to the tools is a grammar. The tool then applies a variety of algorithms to produce a parser that does the right thing. One of the most widely used such tools is yacc, which we'll discuss further in class. See the yacc documentation for some more information about how it works.

Although we don't have time to look at all the techniques that systems like yacc use to build deterministic bottom up parsers, we will look at one of the structures that they can build. A **precedence table** tells us whether to shift or reduce. It uses just two sources of information, the current top of stack symbol and the lookahead symbol. We won't describe how this table is

constructed, but let's look at an example of one and see how it works. For our expression grammar, we can build the following precedence table (where \$ is a special symbol concatenated to the end of each input string that signals the end of the input):

$V \cup \Sigma$	(	)	id	+	*	\$
(						
)		•		•	•	•
id		•		•	•	•
+						
*						
E						
T		•		•		•
F		•		•	•	•

Here's how to read the table. Compare the left most column to the top of the stack and find the row that matches. Now compare the symbols along the top of the chart to the lookahead symbol and find the column that matches. If there's a dot in the corresponding square of the table, then reduce. Otherwise, shift. So let's go back to our example input string  $id + id * id$ . Remember that we had a shift/reduce conflict when the stack's contents were  $T + E$  and the next input symbol was  $*$ . So we look at the next to the last row of the table, the one that has  $T$  as the top of stack symbol. Then we look at the column headed  $*$ . There's no dot, so we don't reduce. But notice that if the lookahead symbol had been  $+$ , we'd have found a dot, telling us to reduce, which is exactly what we'd want to do. Thus this table captures the precedence relationships between the operators  $*$  and  $+$ , plus the fact that we want to associate left when faced with operators of equal precedence.

Deterministic, bottom up parsers of the sort that yacc builds are driven by an even more powerful table called a parse table. Think of the parse table as an extension of the precedence table that contains additional information that has been extracted from an examination of the grammar.

## 7 Closure Properties of Context-Free Languages

**Union:** The CFL's are closed under union. Proof: If  $L_1$  and  $L_2$  are CFL's, then there are, by definition, CFG's  $G_1 = (V_1, \Sigma_1, R_1, S_1)$  and  $G_2 = (V_2, \Sigma_2, R_2, S_2)$  generating  $L_1$  and  $L_2$ , respectively. (Assume that the non-terminal vocabularies of the two grammars are disjoint. We can always rename symbols to achieve this, so there are no accidental interactions between the two rule sets.) Now form CFG  $G = (V_1 \cup V_2, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\}, S)$ .  $G$  generates  $L_1 \cup L_2$  since every derivation from the start symbol of  $G$  must begin either  $S \Rightarrow S_1$  or  $S \Rightarrow S_2$  and thereafter to derive a string generated by  $G_1$  or by  $G_2$ , respectively. Thus all strings in  $L(G_1) \cup L(G_2)$  are generated, and no others.

**Concatenation:** The CFL's are closed under concatenation. The proof is similar. Given  $G_1$  and  $G_2$  as above, form  $G = (V_1 \cup V_2, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \rightarrow S_1 S_2\}, S)$ .  $G$  generates  $L_1 L_2$  since every derivation from  $S$  must begin  $S \Rightarrow S_1 S_2$  and proceed thereafter to derive a string of  $L_1$  concatenated to a string of  $L_2$ . No other strings are produced by  $G$ .

**Kleene star:** The CFL's are closed under Kleene star. Proof: If  $L$  is a CFL, it is generated by some CFG  $G = (V, \Sigma, R, S)$ . Using one new nonterminal symbol  $S'$ , we can form a new CFG  $G' = (V \cup S', \Sigma, R \cup \{S' \rightarrow \epsilon, S' \rightarrow S'S\}, S')$ .  $G'$  generates  $L^*$  since there is a derivation of  $\epsilon$  ( $S' \Rightarrow \epsilon$ ), and there are other derivations of the form  $S' \Rightarrow S'S \Rightarrow S'SS \Rightarrow \dots \Rightarrow S'S \dots SS \Rightarrow S \dots SS$ , which produce finite concatenations of strings of  $L$ .  $G$  generates no other strings.

**Intersection:** The CFL's are *not* closed under intersection. To prove this, it suffices to show one example of two CFL's whose intersection is not context free. Let  $L_1 = \{a^i b^j c^i : i, j \geq 0\}$ .  $L_1$  is context-free since it is generated by a CFG with the rules  $S \rightarrow AC, A \rightarrow aAb, A \rightarrow \epsilon, C \rightarrow cC, C \rightarrow \epsilon$ . Similarly, let  $L_2 = \{a^k b^m c^m : k, m \geq 0\}$ .  $L_2$  is context-free, since it is generated by a CFG similar to the one for  $L_1$ . Now consider  $L_3 = L_1 \cap L_2 = \{a^n b^n c^n : n \geq 0\}$ .  $L_3$  is not context free. We'll prove that in the next section using the context-free pumping lemma. Intuitively,  $L_3$  isn't context free because we can't count a's, b's, and c's all with a single stack.

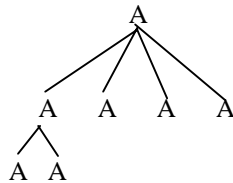
**Intersection with regular languages:** The CFL's are, however, closed under intersection with the regular languages. Given a CFL,  $L$  and a regular language  $R$ , the intersection  $L \cap R$  is a CFL. Proof: Since  $L$  is context-free, there is some non-deterministic PDA accepting it, and since  $R$  is regular, there is some deterministic finite state automaton that accepts it. The two automata can now be merged into a single PDA by a straightforward technique described in class.

**Complementation:** The CFL's are *not* closed under complement. Proof: Since the CFL's are closed under union, if they were also closed under complement, this would imply that they are closed under intersection. This is so because of the set-theoretic equality  $\overline{\overline{L_1} \cup \overline{L_2}} = (L_1 \cap L_2)$ .

## 8 The Context-Free Pumping Lemma

There is a pumping lemma for context-free languages, just as there is one for regular languages. It's a bit more complicated, but we can use it in much the same way to show that some language  $L$  is not in the class of context-free languages. In order to see why the pumping lemma for context-free languages is true, we need to make some observations about parse trees:

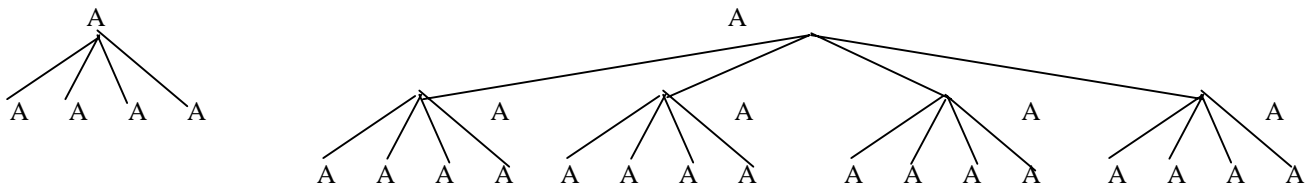
1. A **path** in a parse tree is a continuously descending sequence of connected nodes.
2. The **length** of a path in a parse tree is the number of connections (branches) in it.



3. The **height** of a parse tree is the length of the longest path in it. For example, the parse tree above is of height 2.
4. The **width** of a parse tree is the length of its yield (the string consisting of its leaves). For example, the parse tree above is of width 5.

We observe that in order for a parse tree to achieve a certain width it must attain a certain minimum height. How are height and width connected? The relationship depends on the rules of the grammar generating the parse tree.

Suppose, for example, that a certain CFG contains the rule  $A \rightarrow AAAA$ . Focusing just on derivations involving this rule, we see that a tree of height 1 would have a width of 4. A tree of height 2 would have a *maximum* width of 16 (although there are narrower trees of height 2 of course).



With height 3, the maximum width is 64 (i.e.,  $4^3$ ), and in general a tree of height  $n$  has maximum width of  $4^n$ . Or putting it another way, if a tree is wider than  $4^n$  then it must be of height greater than  $n$ .

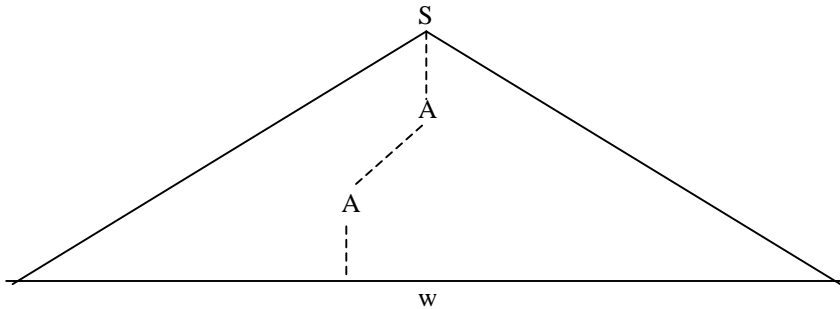
Where does the 4 come from? Obviously from the length of the right-hand side of the rule  $A \rightarrow AAAA$ . If we had started with the rule  $A \rightarrow AAAAAA$ , we would find that a tree of height  $n$  has maximum width  $6^n$ .

What about other rules in the grammar? If it contained both the rules  $A \rightarrow AAAA$  and  $A \rightarrow AAAAAA$ , for example, then the maximum width would be determined by the longer right-hand side. And if there were no other rules whose right-hand sides were longer than 6, then we could confidently say that any parse tree of height  $n$  could be no wider than  $6^n$ .

Let  $p$  = the maximum length of the right-hand side of all the rules in  $G$ . Then any parse tree generated by  $G$  of height  $m$  can

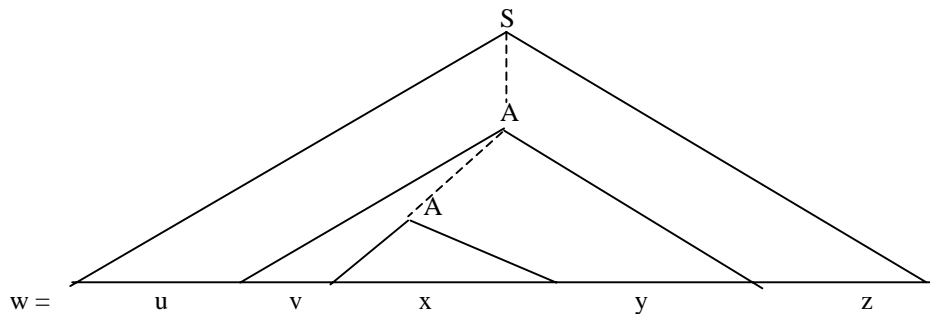
be no wider than  $p^m$ . Equivalently, any parse tree that is generated by  $G$  and that is wider than  $p^m$  must have height greater than  $m$ .

Now suppose we have a CFG  $G = (V, \Sigma, R, S)$ . Let  $n = |V - \Sigma|$ , the size of the non-terminal alphabet. If  $G$  generates a parse tree of width greater than  $p^n$ , then, by the above reasoning, the tree must be of height greater than  $n$ , i.e., it contains a path of length greater than  $n$ . Thus there are more than  $n + 1$  nodes on this path (the number of nodes being one greater than the length of the path), and all of them are non-terminal symbols except possibly the last. Since there are only  $n$  distinct non-terminal symbols in  $G$ , some such symbol must occur more than once along this path (by the Pigeonhole Principle). What this says is that if a CFG generates a long enough string, its parse tree is going to be sufficiently high that it is guaranteed to have a path with some repeated non-terminal symbol along it. Let us represent this situation by the following diagram:

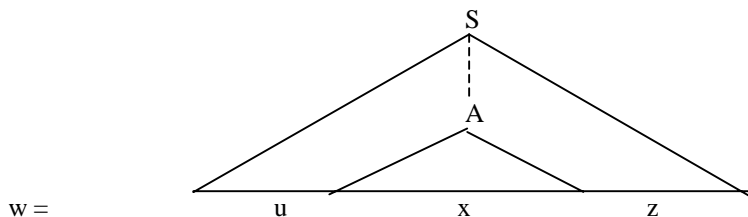


Call the generated string  $w$ . The parse tree has  $S$  as its root, and let  $A$  be a non-terminal symbol (it could be  $S$  itself, of course) that occurs at least twice on some path (indicated by the dotted lines).

Another observation about parse trees: If the leaves are all terminal symbols, then every non-terminal symbol in the tree is the root of a subtree having terminal symbols as its leaves. Thus, the lower instance of  $A$  in the tree above must be the root of a tree with some substring of  $w$  as its leaves. Call this substring  $x$ . The upper instance of  $A$  likewise roots a tree with a string of terminal symbols as its leaves, and furthermore, from the geometry of the tree, we see that this string must include  $x$  as a substring. Call the larger string, therefore,  $vxy$ . This string  $vxy$  is also a substring of the generated string  $w$ , which is to say that for some strings  $u$  and  $z$ ,  $w = uvxyz$ . Attaching these names to the appropriate substrings we have the following diagram:



Now, assuming that such a tree is generated by  $G$  (which will be true on the assumption that  $G$  generates some sufficiently long string), we can conclude that  $G$  must generate some other parse trees as well and therefore their associated terminal strings. For example, the following tree must also be generated:







depending on  $G$  such that, for every string  $w \in L(G)$  where  $|w| > K$ , there are strings  $u, v, x, y, z$  such that

- (1)  $w = uvxyz$ ,
- (2)  $|vy| > 0$ ,
- (3)  $|vxy| \leq M$ , and
- (4) for all  $n \geq 0$ ,  $uv^nxy^n z \in L(G)$ .

Remarks: The constant  $K$  in the lemma is just  $p^n$  referred to above - the length of the longest right-hand side of a rule of  $G$  raised to the power of the number of non-terminal symbols. In applying the lemma we won't care what the value of  $K$  actually is, only that some such constant exists. If  $G$  generates an infinite language, then clearly there will be strings in  $L(G)$  longer than  $K$ , no matter what  $K$  is. If  $L(G)$  is finite, on the other hand, then the lemma still holds (trivially), because  $K$  will have a value greater than the longest strings in  $L(G)$ . So all strings in  $L(G)$  longer than  $K$  are guaranteed to be "pumpable," but no such strings exist, so the lemma is trivially true because the antecedent of the conditional is false. Similarly for  $M$ , which is actually bigger than  $K$ ; it is  $p^{n+1}$ . But, again, all we care about is that if  $L(G)$  is infinite then  $M$  exists. Without knowing what it is, we can describe strings in terms of it and know that we must have pumpable strings.

This lemma, like the pumping lemma for regular languages, addresses the question of how strings grow longer and longer without limit so as to produce infinite languages. In the case of regular languages, we saw that strings grow by repeating some substring any number of times:  $xy^n z \in L$  for all  $n \geq 0$ . When does this happen? Any string in the language of sufficient length is guaranteed to contain such a "pumpable" substring. What length is sufficient? The number of states in the minimum-state deterministic finite state machine that accepts the language. This sets the lower bound for guaranteed pumpability.

For context-free languages, strings grow by repeating two substrings simultaneously:  $uv^nxy^n z \in L$  for all  $n \geq 0$ . This, too, happens when a string in the language is of sufficient length. What is sufficient? Long enough to guarantee that its parse tree contains a repeated non-terminal along some path. Strings this long exceed the lower bound for guaranteed context-free pumpability.

What about the condition that  $|vy| > 0$ , i.e.,  $v$  and  $y$  cannot both be the empty string? This could happen if by the rules of  $G$  we could get from some non-terminal  $A$  back to  $A$  again without producing any terminal symbols in the process, and that's possible with rules like  $A \rightarrow B, B \rightarrow C, C \rightarrow A$ , all perfectly good context-free rules. But given that we have a string  $w$  whose length is greater than or equal to  $K$ , its derivation must have included *some* rules that make the string grow longer; otherwise  $w$  couldn't have gotten as long as it did. Therefore, there must be some path in the derivation tree with a repeated non-terminal that involves branching rules, and along *this* path, at least one of  $v$  or  $y$  is non-empty.

Recall that the corresponding condition for regular languages was  $y \neq \epsilon$ . We justified this by pointing out that if a sufficiently long string  $w$  was accepted by the finite state machine, then there had to be a loop in the machine and that loop must read something besides the empty string; otherwise  $w$  couldn't be as long as it is and still be accepted.

And, finally, what about condition (3),  $|vxy| \leq M$ ? How does this compare to the finite state pumping lemma? The corresponding condition there was that  $|xy| \leq K$ . Since  $|y| \leq |xy|$ , this certainly tells us that the pumpable substring  $y$  is (relatively) short.  $|xy| \leq K$  also tells us that  $y$  occurs close to the beginning of the string  $w = xyz$ . The context-free version, on the other hand, tells us that  $|vxy| \leq M$ , where  $v$  and  $y$  are the pumpable substrings. Since  $|v| \leq |vxy| \leq M$  and  $|y| \leq |vxy| \leq M$ , we know that the pumpable substrings  $v$  and  $y$  are short. Furthermore, from  $|vxy| \leq M$ , we know that  $v$  and  $y$  must occur close to each other (or at least not arbitrarily far away from each other). Unlike in the regular pumping lemma, though, they do not necessarily occur close to the beginning of the string  $w = uvxyz$ . This is the reason that context-free pumping lemma proofs tend to have more cases: the  $v$  and  $y$  pumpable substrings can occur anywhere within the string  $w$ .

Note that this Pumping Lemma, like the one for regular languages, is an if-then statement not an iff statement. Therefore, it cannot be used to show that a language *is* context-free, only that it is *not*.

**Example 1:** Show that  $L = \{a^n b^n c^n : n \geq 0\}$  is not context-free.

If  $L$  were context-free (i.e., if there were a context-free grammar generating  $L$ ), then the Pumping Lemma would apply. Then

there would be a constant  $K$  such that every string in  $L$  of length greater than  $K$  would be "pumpable." We show that this is not so by exhibiting a string  $w$  in  $L$  that is of length greater than  $K$  and that is not pumpable. Since we want to rely on clause 3 of the pumping lemma, and it relies on  $M > K$ , we'll actually choose  $w$  in terms of  $M$ .

Let  $w = a^M b^M c^M$ . (Note that this is a *particular string*, not a language or a variable expression for a string.  $M$  is some number whose exact value we don't happen to know; it might be 23, for example. If so,  $w$  would be the unique string  $a^{23} b^{23} c^{23}$ .) This string is of length greater than  $K$  (of length  $3M$ , where  $M$  is greater than  $K$ , in fact), and it is a string in the language  $\{a^n b^n c^n : n \geq 0\}$ . Therefore, it satisfies the criteria for a pumpable string according to the Pumping Lemma--provided, of course, that  $L$  is context-free.

What does it mean to say that  $a^M b^M c^M$  is pumpable? It means that there is *some* way to factor this string into five parts -  $u, v, x, y, z$  - meeting the following conditions:

1.  $v$  and  $y$  are not both the empty string (although any of  $u$ ,  $x$ , or  $z$  could be empty),
2.  $|vxy| \leq M$ ,
3.  $uxz \in L$ ,  $uvxyz \in L$ ,  $uvvxyyz \in L$ ,  $uvvvxyyyz \in L$ , etc.; i.e., for all  $n \geq 0$ ,  $uv^n xy^n z \in L$ .

We now show that there is *no way* to factor  $a^M b^M c^M$  into 5 parts meeting these conditions; thus,  $a^M b^M c^M$  is *not* a pumpable string, contrary to the stipulation of the Pumping Lemma, and thus  $L$  is *not* a context-free language.

How do we do this? We show that no matter how we try to divide  $a^M b^M c^M$  in ways that meet the first two conditions, the third condition always falls. In other words, every "legal" division of  $a^M b^M c^M$  fails to be pumpable--that is, there is some value of  $n$  for which  $uv^n xy^n z \notin L$ .

There are clearly a lot of ways to divide this string into 5 parts, but we can simplify the task by grouping the divisions into cases just as we did with the regular language Pumping Lemma:

*Case 1:* Either  $v$  or  $y$  consists of more than different letter (e.g.,  $aab$ ). No such division is pumpable, since for any  $n \geq 2$ ,  $uv^n xy^n z$  will contain some letters not in the correct order to be in  $L$ . Now that we've eliminated this possibility, all the remaining cases can assume that both  $v$  and  $y$  contain only  $a$ 's, only  $b$ 's, or only  $c$ 's (although one could also be  $\epsilon$ ).

*Case 2:* Both  $v$  and  $y$  are located within  $a^M$ . No such division is pumpable, since we will pump in only  $a$ 's. So, for  $n \geq 2$ ,  $uv^n xy^n z$  will contain more  $a$ 's than  $b$ 's or  $c$ 's and therefore won't be in  $L$ . (Note that  $n = 0$  also works.)

*Cases 3, 4:* Both  $v$  and  $y$  are located within  $b^M$  or  $c^M$ . No such division is pumpable, by the same logic as in Case 2.

*Case 5:*  $v$  is located within  $a^M$  and  $y$  is located within  $b^M$ . No such division is pumpable, since for  $n \geq 2$ ,  $uv^n xy^n z$  will contain more  $a$ 's than  $c$ 's or more  $b$ 's than  $c$ 's (or both) and therefore won't be in  $L$ . ( $n = 0$  also works here.)

*Cases 6, 7:*  $v$  is located within  $a^M$  and  $y$  is located within  $c^M$ , or  $v$  is located within  $b^M$  and  $y$  is located within  $c^M$ . No such division is pumpable, by the same logic as in Case 5.

Since every way of dividing  $a^M b^M c^M$  into 5 parts (such that the 2nd and 4th are not both empty) is covered by (at least one of) the above 7 cases, and in each case we find that the resulting division is not pumpable, we conclude that there is *no* division of  $a^M b^M c^M$  that is pumpable. Since all this was predicated on the assumption that  $L$  was a context-free language, we conclude that  $L$  is not context-free after all.

Notice that we didn't need to use condition the fact that  $|vxy|$  must be less than  $M$  in this proof, although we could have used it as an alternative way to handle case 6, since it prevents  $v$  and  $y$  from being separated by a region of size  $M$ , which is exactly the size of the region of  $b$ 's that occurs between the  $a$ 's and the  $c$ 's.

**Example 2:** Show that  $L = \{w \in \{a, b, c\}^* \mid \#(a, w) = \#(b, w) = \#(c, w)\}$  is not context free. (We use the notation  $\#(a, w)$  to mean the number of  $a$ 's in the string  $w$ .)

Let's first try to use the pumping lemma. We could again choose  $w = a^M b^M c^M$ . But now we can't immediately brush off case 1 as we did in Example 1, since  $L$  allows for strings that have the  $a$ 's,  $b$ 's, and  $c$ 's interleaved. In fact, this time there *are* ways to divide  $a^M b^M c^M$  into 5 parts ( $v, y$  not both empty), such that the result is pumpable. For example, if  $v$  were  $ab$  and  $y$  were  $c$ , then  $uv^n xy^n z$  would be in  $L$  for all  $n \geq 0$ , since it would still contain equal numbers of  $a$ 's,  $b$ 's, and  $c$ 's.

So we need some additional help, which we'll find in the closure theorems for context-free languages. Our problem is that the definition of  $L$  is too loose, so it's too easy for the strings that result from pumping to meet the requirements for being in  $L$ . We need to make  $L$  more restrictive. Intersection with another language would be one way we could do that. Of course, since the context-free languages are not closed under intersection, we can't simply consider some new language  $L' = L \cap L_2$ , where  $L_2$  is some arbitrary context-free language. Even if we could use pumping to show that  $L'$  isn't context free, we'd know nothing about  $L$ . But the context-free languages *are* closed under intersection with regular languages. So if we construct a new language  $L' = L \cap L_2$ , where  $L_2$  is some arbitrary *regular* language, and then show that  $L'$  is not context free, we know that  $L$  isn't either (since, if it were, its intersection with a regular language would also have to be context free). Generally in problems of this sort, the thing to do is to use intersection with a regular language to constrain  $L$  so that all the strings in it must have identifiable regions of symbols. So what we want to do here is to let  $L_2 = a^*b^*c^*$ . Then  $L' = L \cap L_2 = a^n b^n c^n$ . If we hadn't just proved that  $a^n b^n c^n$  isn't context free, we could easily do so. In either case, we know immediately that  $L$  isn't context free.

# Recursively Enumerable Languages, Turing Machines, and Decidability

## 1 Problem Reduction: Basic Concepts and Analogies

The concept of problem reduction is simple at a high level. You simply take an algorithm that solves one problem and use it as a subroutine to solve another problem. For example, suppose we have two TM's:  $C$ , which turns  $\square w \square$  into  $\square w \square w \square$  and  $S_L$ , which turns  $\dots \square w \square$  into  $\dots w \square \square$  (i.e., it shifts  $w$  one square to the left). Then we can build a TM  $M'$  that computes  $f(w) = ww$  by simply letting  $M' = CS_L$ . We have reduced the problem of computing  $f$  to the problem of copying a string and then shifting it.

Let's consider another example. Suppose we had a function  $\text{sqr}(m: \text{integer}): \text{integer}$ , which accepts an integer input  $m$  and returns  $m^2$ . Then we could write the following function to compute  $g(m) = m^2 + 2m + 1$ :

```
function g(m: integer): integer;
begin
    return sqr(m + 1);
end;
```

We have reduced the problem of computing  $m^2 + 2m + 1$  to the problem of computing  $m + 1$  and squaring it.

We are going to be using reduction specifically for decision problems. In other words, we're interested in a particular class of boolean functions whose job is to look at strings and tell us, for each string, whether or not it is in the language we are concerned with. For example, suppose we had a TM  $M$  that decides the language  $a^*b^+$ . Then we could make a new TM  $M'$  to decide  $a^*b^*$ :  $M'$  would find the first blank to the right of its input and write a single  $b$  there. It would then move its read head back to the beginning of the string and invoke  $M$  on the tape. Why does this work? Because  $x \in a^*b^*$  iff  $xb \in a^*b^+$ . Looking at this in the more familiar procedural programming style, we are saying that if we have a function:  $f1(x: \text{string}): \text{boolean}$ , which tells us whether  $x \in a^*b^+$ , then we could write the following function that will correctly tell us whether  $x \in a^*b^*$ .

```
function f2(x: string): boolean;
begin
    return f1(x || 'b');
end;
```

If, for some reason, you believed that  $a^*b^*$  were an undecidable language, then this reduction would force you to believe that  $a^*b^+$  is also undecidable. Why? Because we have shown that we can decide  $a^*b^*$  provided only that  $f1$  does in fact decide  $a^*b^+$ . If we know that we can't decide  $a^*b^*$ , there must be something standing in the way. Unless we're prepared to believe that subroutine invocation is not computable or that concatenation of a single character to the end of a string is not computable, we must assign blame to  $f1$  and conclude that we didn't actually have a correct  $f1$  program to begin with.

These examples should all make sense. The underlying concept is simple. Things will become complicated in a bit because we will begin considering TM descriptions as the input strings about which we want to ask questions, rather than simple strings of  $a$ 's and  $b$ 's.

Sometimes, we'll use a slightly different but equivalent way of asking our question. Rather than asking whether a language is decidable, we may ask whether a problem is solvable. When we say that a problem is unsolvable, what we mean is that the corresponding language is undecidable. For example, consider the problem of determining, given a TM  $M$  and string  $w$ , whether or not  $M$  accepts  $w$ . We can ask whether this problem is solvable. But notice that this same problem can be phrased a language recognition problem because it is equivalent to being able to decide the language:

$$H = \{ "M" "w" : w \in L(M) \}.$$

Read this as:  $H$  is the language that consists of all strings that can be formed from two parts: the encoding of a Turing

Machine  $M$ , followed by the encoding of a string  $w$  (which we can think of as the input to  $M$ ), with the additional constraint that TM  $M$  halts on input  $w$  (which is equivalent to saying that  $w$  is in the language accepted by  $M$ ).

“Solving a problem” is the higher level notion, which is commonly used in the programming/algorithm context. In our theoretical framework, we use the term “deciding a language” because we are talking about Turing Machines, which operate on strings, and we have a carefully constructed theory that lets us talk about Turing Machines as language recognizers.

In the following section, we’ll go through several examples in which we use the technique of problem reduction to show that some new problem is unsolvable (or, alternatively, that the corresponding language is undecidable). All of these proofs depend ultimately on our proof, using diagonalization, of the undecidability of the halting problem ( $H$  above).

## 2 Some Reductions Presented in Gory Detail

**Example 1:** Given a TM  $M$ , does  $M$  halt on input  $\epsilon$ ? (i.e., given  $M$ , is  $\epsilon \in L(M)$ ?) This problem is undecidable because we can reduce the Halting Problem  $H$  to it. What this means is that an algorithm to answer this question could be used as a subroutine in an algorithm (which is otherwise clearly effective) to solve the Halting problem. But we know the Halting problem is unsolvable; therefore this question is unsolvable. So how do we prove this?

First we’ll prove this using the TM/language framework. In other words, we’re going to show that the following language  $LE$  is not decidable:

$$LE = \{ "M" : \epsilon \in L(M) \}$$

We will show that if  $LE$  is decidable, so is  $H = \{ "M" "w" : w \in L(M) \}$ .

Suppose  $LE$  is decidable; then some TM  $M_{LE}$  decides it. We can now show how to construct a new Turing Machine  $M_H$ , which will invoke  $M_{LE}$  as a subroutine, and which will decide  $H$ . In the process of doing so, we’ll use only clearly computable functions (other than  $M_{LE}$ , of course). So when we finish and realize that we have a contradiction (since we know that  $M_H$  can’t exist), we know that the blame must rest on  $M_{LE}$  and thus we know that  $M_{LE}$  cannot exist.

$M_H$  is the Turing Machine that operates as follows on the inputs “ $M$ ”, “ $w$ ”:

1. Construct a new TM  $M^*$ , which behaves as follows:
  - 1.1. Copy “ $w$ ” onto its tape.
  - 1.2. Execute  $M$  on the resulting tape.
2. Invoke  $M_{LE}(M^*)$ .

If  $M_{LE}$  returns True, then we know that  $M$  (the original input to  $M_H$ ) halts on  $w$ . If  $M_{LE}$  returns False, then we know that it doesn’t. Thus we have built a supposedly unbuildable  $M_H$ . How did we do it? We claimed when we asserted the existence of  $M_{LE}$  that we could answer what appears to be a more limited question, does  $M$  halt on the empty tape? But we can find out whether  $M$  halts on any other specific input ( $w$ ) by constructing a machine ( $M^*$ ) that starts by writing  $w$  on top of whatever was originally on its tape (thus it ignores its actual input) and then proceeds to do whatever  $M$  would have done. Thus  $M^*$  behaves the same on all inputs. Thus if we knew what it does on any one input, we’d know what it does for all inputs. So we ask  $M_{LE}$  what it does on the empty string. And that tells us what it does all the time, which must be, by the way we constructed it, whatever the original machine  $M$  does on  $w$ .

The only slightly tricky thing here is the procedure for constructing  $M^*$ . Are we sure that it is in fact computable? Maybe we’ve reached the contradiction of claiming to have a machine to solve  $H$  not by erroneously claiming to have  $M_{LE}$  but rather by erroneously claiming to have a procedure for constructing  $M^*$ . But that’s not the case. Why not? It’s easy to see how to write a procedure that takes a string  $w$  and builds  $M^*$ . For example, if “ $w$ ” is “ $ab$ ”, then  $M^*$  must be:

ERaRbL<sub>□</sub>M, where  $E$  is a TM that erases its tape and then moves the read head back to the first square.

In other words, we erase the tape, move back to the left, then move right one square (leaving one blank ahead of the new tape contents), write  $a$ , move right, write  $b$ , move left until we get back to the blank that’s just to the left of the input, and then execute  $M$ .

The Turing Machine to construct  $M^*$  is a bit too complicated to write here, but we can see how it works by describing it in a more standard procedural way: It first writes  $ER$ . Then, for each character in  $w$ , it writes that character and  $R$ . Finally it writes  $L_{\sqcup}M$ .

To make this whole process even clearer, let's look at this problem not from the point of view of the decidability of the language  $H$  but rather by asking whether we can solve the Halting problem. To do this, let's describe in standard code how we could solve the Halting problem if we had a subroutine  $M_{LE}(M: TM)$  that could tell us whether a particular Turing Machine halts on the empty string. We'll assume a datatype  $TM$ . If you want to, you can think of objects of this type as essentially strings that correspond to valid Turing Machines. It's like thinking of a type  $C$  program, which is all the strings that are valid  $C$  programs.

We can solve the Halting program with following function `Halts`:

Function `Halts(M: TM, w: string): Boolean;`

```

M* := Construct(M, w);
Return MLE(M*);
end;
```

Function `Construct(M: TM, w: string): TM;`

```

/* Construct builds a machine that first erases its tape. Then it copies w onto its tape and moves its
/* read head back to the left ready to begin reading w. Finally, it executes M.
Construct := Erase; /* Erase is a string that corresponds to the TM that erases its input tape.
For each character c in w do
    Construct := Construct || "R" || c;
end;
Construct := Construct || L_{\sqcup}M;
Return(Construct);
```

Function `MLE(M: TM): Boolean;`

The function we claim tells us whether  $M$  halts on the empty string.

The most common mistake that people make when they're trying to use reduction to show that a problem isn't solvable (or that a language isn't decidable) is to do the reduction backwards. In this case, that would mean we would put forward the following argument: "Suppose we had a program `Halts` to solve the Halting problem (the general problem of determining whether a  $TM$   $M$  halts on some arbitrary input  $w$ ). Then we could use it as a subroutine to solve the specific problem of determining whether a  $TM$   $M$  halts on the empty string. We'd simply invoke `Halts` and pass it  $M$  and  $\epsilon$ . If `Halts` returns `True`, then we say yes. If `Halts` returns `False`, we say no. But since we know that `Halts` can't exist, no solution to our problem can exist either." The flaw in this argument is the last sentence. Clearly, since `Halts` can't exist, this particular approach to solving our problem won't work. But this says nothing about whether there might be some other way to solve our problem.

To see this flaw even more clearly, let's consider applying it in a clearly ridiculous way: "Suppose we had a program `Halts` to solve the Halting problem. Then we could use it as a subroutine to solve the problem of adding two numbers  $a$  and  $b$ . We'd simply invoke `Halts` and pass it the trivial  $TM$  that simply halts immediately and the input  $\epsilon$ . If `Halts` returns `True`, then we return  $a+b$ . If `Halts` returns `False`, we also return  $a+b$ . But since we know that `Halts` can't exist, no solution to our problem can exist either." Just as before, we have certainly written a procedure for adding two numbers that won't work, since `Halts` can't exist. But there are clearly other ways to solve our problem. We don't need `Halts`. That's totally obvious here. It's less so in the case of attempting to build  $M_{LE}$ . But the logic is the same in both cases: flawed.

**Example 2:** Given a  $TM$   $M$ , is  $L(M) \neq \emptyset$ ? (In other words, does  $M$  halt on anything at all?). Let's do this one first using the solvability of the problem perspective. Then we'll do it from the decidability of the language point of view.

This time, we claim that there exists:

Function `MLA(M: TM): Boolean;`

Returns `T` if  $M$  halts on any inputs at all and `False` otherwise.

We show that if this claim is true and MLA does in fact exist, then we can build a function MLE that solves the problem of determining whether a TM accepts the empty string. We already know, from our proof in Example 1, that this problem isn't solvable. So if we can do it using MLA (and some set of clearly computable functions), we know that MLA cannot in fact exist.

The reduction we need for this example is simple. We claim we have a machine MLA that tells us whether some machine M accepts anything at all. If we care about some particular input to M (for example, we care about  $\epsilon$ ), then we will build a new machine  $M^*$  that erases whatever was originally on its tape. Then it copies onto its tape the input we care about (i.e.,  $\epsilon$ ) and runs M. Clearly this new machine  $M^*$  is oblivious to its actual input. It either always accepts (if M accepts  $\epsilon$ ) or always rejects (if M rejects  $\epsilon$ ). It accepts everything or nothing. So what happens if we pass  $M^*$  to MLA? If  $M^*$  always accepts, then its language is not the empty set and MLA will return True. This will happen precisely in case M halts on  $\epsilon$ . If  $M^*$  always rejects, then its language is the empty set and MLA will return False. This will happen precisely in case M doesn't halt on  $\epsilon$ . Thus, if MLA really does exist, we have a way to find out whether any machine M halts on  $\epsilon$ :

```
Function MLE(M: TM): Boolean;
  M* := Construct(M);
  Return MLA(M*);
end;
```

```
Function Construct(M: TM): TM; /* This time, we build an M* that simply erases its input and then runs M
                               /*(thus running M on  $\epsilon$ ).
  Construct := Erase; /* Erase is a string that corresponds to the TM that erases its input tape.
  Construct := Construct || M;
  Return(Construct);
```

But we know that MLE can't exist. Since everything in its code, with the exception of MLA, is trivially computable, the only way it can't exist is if MLA doesn't actually exist. Thus we've shown that the problem determining whether or not a TM M halts on any inputs at all isn't solvable.

Notice that this argument only works because everything else that is done, both in MLE and in Construct is clearly computable. We could write it all out in the Turing Machine notation, but we don't need to, since it's possible to prove that anything that can be done in any standard programming language can also be done with a Turing Machine. So the fact that we can write code for it is good enough.

Whenever we want to try to use this approach to decide whether or not some new problem is solvable, we can choose to reduce to the new problem any problem that we already know to be unsolvable. Initially, the only problem we knew wasn't solvable was the Halting problem, which we showed to be unsolvable using diagonalization. But once we have used reduction to show that other problems aren't solvable either, we can use any of them for our next problem. The choice is up to you. Whatever is easiest is the thing to use.

When we choose to use the problem solvability perspective, there is always a risk that we may make a mistake because we haven't been completely rigorous in our definition of the mechanisms that we can use to solve problems. One big reason for even defining the Turing Machine formalism is that it is both simple and rigorously defined. Thus, although the problem solvability perspective may seem more natural to us, the language decidability perspective gives us a better way to construct rigorous proofs.

So let's show that the language

$LA = \{ "M": L(M) \neq \emptyset \}$  is undecidable.

We will show that if LA were decidable, then  $LE = \{ "M": \epsilon \in L(M) \}$  would also be decidable. But of course, we know that it isn't.

Suppose LA is decidable; then some TM  $M_{LA}$  decides it. We can now show how to construct a new Turing Machine  $M_{LE}$ ,



which will invoke  $M_{LA}$  as a subroutine, and which will decide LE:

$M_{LE}(M)$ : /\* A decision procedure for  $LE = \{ "M": \epsilon \in L(M) \}$

1. Construct a new TM  $M^*$ , which behaves as follows:
  - 1.1. Erase its tape.
  - 1.2. Execute  $M$  on the resulting empty tape.
2. Invoke  $M_{LA}(M^*)$ .

It's clear that  $M_{LE}$  effectively decides LE (if  $M_{LA}$  really exists). Why?  $M_{LE}$  returns True iff  $M_{LA}$  returns True. That happens, by its definition, if it is passed a TM that accepts at least one string. We pass it  $M^*$ .  $M^*$  accepts at least one string (in fact, it accepts all strings) precisely in case  $M$  accepts the empty string. If  $M$  does not accept the empty string, then  $M^*$  accepts nothing and  $M_{LE}$  returns False.

**Example 3:** Given a TM  $M$ , is  $L(M) = \Sigma^*$ ? (In other words, does  $M$  accept everything?). We can show that this problem is unsolvable by using almost exactly the same technique we just used. In example 2, we wanted to know whether a TM accepted anything at all. Now we want to know whether it accepts everything. We will answer this question by showing that the language

$$L\Sigma = \{ "M": L(M) = \Sigma^* \} \text{ is undecidable.}$$

Recall the machine  $M^*$  that we constructed for Example 2. It erases its tape and then runs  $M$  on the empty tape. Clearly  $M^*$  either accepts nothing or it accepts everything, since its behavior is independent of its input.  $M^*$  is exactly what we need for this proof too. Again we'll choose to reduce the language  $LE = \{ "M": \epsilon \in L(M) \}$  to our new problem  $L$ :

If  $L\Sigma$  is decidable, then there's a TM  $M_{L\Sigma}$  that decides it. In other words, there's a TM that tells us whether or not some other machine  $M$  accepts everything. If  $M_{L\Sigma}$  exists, then we can define the following TM to decide LE:

$M_{LE}(M)$ : /\* A decision procedure for  $LE = \{ "M": \epsilon \in L(M) \}$

1. Construct a new TM  $M^*$ , which behaves as follows:
  - 1.1. Erase its tape.
  - 1.2. Execute  $M$  on the resulting empty tape.
2. Invoke  $M_{L\Sigma}(M^*)$ .

Step 2 will return True if  $M^*$  halts on all strings in  $\Sigma^*$  and False otherwise. So it will return True if and only  $M$  halts on  $\epsilon$ . This would seem to be a correct decision procedure for LE. But we know that such a procedure cannot exist and the only possible flaw in the procedure we've given is  $M_{L\Sigma}$ . So  $M_{L\Sigma}$  doesn't exist either.

**Example 4:** Given a TM  $M$ , is  $L(M)$  infinite? Again, we can use  $M^*$ . Remember that  $M^*$  either halts on nothing or it halts on all elements of  $\Sigma^*$ . Assuming that  $\Sigma \neq \emptyset$ , that means that  $M^*$  either halts on nothing or it halts on an infinite number of strings. It halts on everything if its input machine  $M$  halts on  $\epsilon$ . Otherwise it halts on nothing. So we can show that the language

$$LI = \{ "M": L(M) \text{ is infinite} \}$$

is undecidable by reducing the language  $LE = \{ "M": \epsilon \in L(M) \}$  to it.

If  $LI$  is decidable, then there is a Turing Machine  $M_{LI}$  that decides it. Given  $M_{LI}$ , we decide LE as follows:

$M_{LE}(M)$ : /\* A decision procedure for  $LE = \{ "M": \epsilon \in L(M) \}$

1. Construct a new TM  $M^*$ , which behaves as follows:
  - 1.1. Erase its tape.
  - 1.2. Execute  $M$  on the resulting empty tape.
2. Invoke  $M_{LI}(M^*)$ .

Step 2 will return True if  $M^*$  halts on an infinite number of strings and False otherwise. So it will return True if and only  $M$  halts on  $\epsilon$ .

This idea that a single construction may be the basis for several reduction proofs is important. It derives from the fact that several quite different looking problems may in fact be distinguishing between the same two cases.

**Example 5:** Given two TMs,  $M_1$  and  $M_2$ , is  $L(M_1) = L(M_2)$ ? In other words, is the language  $LEQ = \{ \langle M_1 \rangle \langle M_2 \rangle : L(M_1) = L(M_2) \}$  decidable?

Now, for the first time, we want to answer a question about the relationship of two Turing Machines to each other. How can we solve this problem by reducing to it any of the problems we already know to be undecidable? They all involve only a single machine. The trick is to use a constant, a machine whose behavior we are certain of. So we define  $M\#$ , which halts on all inputs.  $M\#$  is trivial. It ignores its input and goes immediately to a halt state.

If  $LEQ$  is decidable, then there is a TM  $M_{LEQ}$  that decides it. Using  $M_{LEQ}$  and  $M\#$ , we can decide the language  $L\Sigma = \{ \langle M \rangle : L(M) = \Sigma^* \}$  (which we showed in example 3 isn't decidable) as follows:

$M_{L\Sigma}(M)$ : /\* A decision procedure for  $L\Sigma$

1. Invoke  $M_{LEQ}(M, M\#)$ .

Clearly  $M$  accepts everything if it is equivalent to  $M\#$ , which is exactly what  $M_{LEQ}$  tells us.

This reduction is an example of an easy one. To solve the unsolvable problem, we simply pass the input directly into the subroutine that we are assuming exists, along with some simple constant. We don't need to do any clever constructions. The reason this was so simple is that our current problem  $LEQ$ , is really just a generalization of a more specific problem we've already shown to be unsolvable. Clearly if we can't solve the special case (determining whether a machine is equivalent to  $M\#$ ), we can't solve the more general problem (determining whether two arbitrary machines are equivalent).

**Example 6:** Given a TM  $M$ , is  $L(M)$  regular? Alternatively, is  $LR = \{ \langle M \rangle : L(M) \text{ is regular} \}$  decidable?

To answer this one, we'll again need to use an interesting construction. To do this, we'll make use of the language

$$H = \{ \langle M \rangle \langle w \rangle : w \in L(M) \}$$

Recall that  $H$  is just the set of strings that correspond to a (TM, input) pair, where the TM halts on the input.  $H$  is not decidable (that's what we proved by diagonalization). But it is semidecidable. We can easily build a TM  $H_{\text{semi}}$  that halts whenever the TM  $M$  halts on input  $w$  and that fails to halt whenever  $M$  doesn't halt on  $w$ . All  $H_{\text{semi}}$  has to do is to simulate the execution of  $M$  on  $w$ . Note also that  $H$  isn't regular (which we can show using the pumping theorem).

Suppose that, from  $M$  and  $H_{\text{semi}}$ , we construct a machine  $M\$$  that behaves as follows: given an input string  $w$ , it first runs  $H_{\text{semi}}$  on  $w$ . Clearly, if  $H_{\text{semi}}$  fails to halt on  $w$ ,  $M\$$  will also fail to halt. But if  $H_{\text{semi}}$  halts, then we move to the next step, which is to run  $M$  on  $\epsilon$ . If we make it here, then  $M\$$  will halt precisely in case  $M$  would halt on  $\epsilon$ . So our new machine  $M\$$  will either:

1. Accept  $H$ , which it will do if  $\epsilon \in L(M)$ , or
2. Accept  $\emptyset$ , which it will do if  $\epsilon \notin L(M)$ .

Thus we see that  $M\$$  will accept either

1. A non regular language,  $H$ , which it will do if  $\epsilon \in L(M)$ , or
2. A regular language  $\emptyset$ , which it will do if  $\epsilon \notin L(M)$ .

So, if we could tell whether  $M\$$  accepts a regular language or not, we'd know whether or not  $M$  accepts  $\epsilon$ .

We're now ready to show that  $LR$  isn't decidable. If it were, then there would be some TM  $M_{LR}$  that decided it. But  $M_{LR}$  cannot exist, because, if it did, we could reduce  $LE = \{ \langle M \rangle : \epsilon \in L(M) \}$  to it as follows:

$M_{LE}(M)$ : /\* A decision procedure for  $LE = \{ \langle M \rangle : \epsilon \in L(M) \}$

1. Construct a new TM  $M\$(w)$ , which behaves as follows:
  - 1.1. Execute  $H_{\text{semi}}$  on  $w$ .

- 1.2. Execute  $M$  on  $\epsilon$ .
2. Invoke  $M_{LR}(M\$)$ .
3. If the result of step 2 is True, return False; if the result of step 2 is False, return True.

$M_{LE}$ , as just defined, effectively decides LE. Why? If  $\epsilon \in L(M)$ , then  $L(M\$)$  is  $H$ , which isn't regular, so  $M_{LR}$  will say False and we will, correctly, say True. If  $\epsilon \notin L$ , then  $L(M\$)$  is  $\emptyset$ , which is regular, so  $M_{LR}$  will say True and we will, correctly, say False.

By the way, we can use exactly this same argument to show that  $LC = \{ \langle M \rangle : L(M) \text{ is context free} \}$  and  $LD = \{ \langle M \rangle : L(M) \text{ is recursive} \}$  are undecidable. All we have to do is to show that  $H$  is not context free (by pumping) and that it is not recursive (which we did with diagonalization).

**Example 7:** Given a TM  $M$  and state  $q$ , is there any configuration  $(p, uqv)$ , with  $p \neq q$ , that yields a configuration whose state is  $q$ ? In other words, is there any state  $p$  that could possibly lead  $M$  to  $q$ ? Unlike many (most) of the questions we ask about Turing Machines, this one is not about future behavior. (e.g., "Will the Turing Machine do such and such when started from here?") So we're probably not even tempted to try simulation (which rarely works anyway).

But there is a way to solve this problem. In essence, we don't need to consider the infinite number of possible configurations of  $M$ . All we need to do is to examine the (finite) transition table of  $M$  to see whether there is any transition from some state other than  $q$  (call it  $p$ ) to  $q$ . If there is such a transition (i.e., if  $\exists p, \sigma, \tau$  such that  $\delta(p, \sigma) = (q, \tau)$ ), then the answer is yes. Otherwise, the answer is no.

### 3 Rice's Theorem

Rice's Theorem makes a very general statement about an entire class of languages that are not recursive. Thus, for some languages, it is an alternative to problem reduction as a technique for proving that a language is not recursive.

There are several different forms in which Rice's Theorem can be stated. We'll present two of them here:

(Form 1) Any nontrivial property of the recursively enumerable languages is not decidable.

(Form 2) Suppose that  $C$  is a proper, nonempty subset of the class of all recursively enumerable languages. Then the following language is undecidable:  $LC = \{ \langle M \rangle : L(M) \text{ is an element of } C \}$ .

These two statements look quite different but are in fact nearly equivalent. (Although Form 1 is stronger since it makes a claim about the language, no matter how you choose to define the language. So it applies given a grammar, for example. Form 2 only applies directly if the way you define the language is by a Turing Machine that semidecides it. But even this difference doesn't really matter since we have algorithms for constructing a Turing Machine from a grammar and vice versa. So it would just take one more step if we started with a grammar and wanted to use Form 2). But, if we want to prove that a language of Turing machine descriptions is not recursive using Rice's Theorem, you must do the same things, whichever description of it you prefer.

We'll consider Form 1 first. To use it, we first, we have to guarantee that the property we are concerned with is a property (predicate) whose domain is the set of recursively enumerable languages. Here are some properties  $P$  whose domains are the RE languages:

- 1)  $P$  is true of any RE language that contains an even number of strings and false of any RE language that contains an odd number of strings.
- 2)  $P$  is true of any RE language that contains all strings over its alphabet and false for all RE languages that are missing any strings over their alphabet.
- 3)  $P$  is true of any RE language that is empty (i.e., contains no strings) and false of any RE language that contains any strings.
- 4)  $P$  is true of any RE language

- 5) P is true of any RE language that can be semidecided by a TM with an even number of states and false for any RE language that cannot be semidecided by such a TM.
- 6) P is true of any RE language that contains at least one string of infinite length and false of any RE language that contains no infinite strings.

Here are some properties whose domains are not the RE languages:

- 1') P is true of Turing machines whose first move is to write "a" and false of other Turing machines.
- 2') P is true of two tape Turing machines and false of all other Turing machines.
- 3') P is true of the negative numbers and false of zero and the positive numbers.
- 4') P is true of even length strings and false of odd length strings.

We can attempt to use Rice's Theorem to tell us that properties in the first list are undecidable. It won't help us at all for properties in the second list.

But now we need to do one more thing: We must show that P is a *nontrivial* property. Any property P is nontrivial if it is not equivalent to True or False. In other words, it must be true of at least one language and false of at least one language.

Let's look at properties 1-6 above:

- 1) P is true of {"a", "b"} and false of {"a"}, so it is nontrivial.
- 2) Let's just consider the case in which  $\Sigma$  is {a, b}. P is true of  $\Sigma^*$  and false of {"a"}.
- 3) P is true of  $\emptyset$  and P is false of every other RE language.
- 4) P is true of any RE language and false of nothing, so P is trivial.
- 5) P is true of any RE language and false of nothing, so P is trivial. Why? Because, for any RE language L there exists a semideciding TM M. If M has an even number of states, then P is clearly true. If M has an odd number of states, then create a new machine M' identical to M except it has one more state. This state has no effect on M's behavior because there are no transitions in to it. But it guarantees that M' has an even number of states. Since M' accepts L (because M does), P is true of L. So P is true of all RE languages.
- 6) P is false for all RE languages. Why? Because the definition of a language is a set of strings, each of finite length. So no RE language contains a string of infinite length.

So we can use Rice's theorem to prove that the set of RE languages possessing any one of properties 1, 2, or 3 is not recursive. But it does not tell us anything about the set of RE languages possessing property 3, 4, or 5.

In summary, to apply this version of Rice's theorem, it is necessary to do three things:

- 0) Specify a property P.
- 1) Show that the domain of P is the set of recursively enumerable languages.
- 2) Show that P is nontrivial by showing:
  - a) That P is true of at least one language, and
  - b) That P is false of at least on language.

Now let's try to use Form 2. We must find a C that is a proper, nonempty subset of the class of all recursively enumerable language.

First we notice that this version is stated in terms of C, a subset of the RE languages, rather than P, a property (predicate) that is true of the RE languages. But this is an insignificant difference. Given a universe U, then one way to define a subset S of U is by a characteristic function that, for any candidate element x, returns true if  $x \in S$  and false otherwise. So the P that corresponds to S is simply whatever property the characteristic function tests for. Alternatively, for any subset S there must exist a characteristic function for S (although that function need not be computable – that's a different issue.) So given a set S, we can define the property P as "is a member of S." or "possesses whatever property it takes to be determined to be n S." So Form 2 is stated in terms of the set of languages that satisfy some property P instead of being stated in terms of P directly, but as there is only one such set for any property P and there is only one such property P (viewed simply as a truth table, ignoring how you say it in English) for any set, it doesn't matter which specification we use.

Next we notice that this version requires that  $C$  be a proper, nonempty subset of the class of RE languages. But this is exactly the same as requiring that  $P$  be nontrivial. Why? For  $P$  to be nontrivial, then there must exist at least one language of which it is true and one of which it is false. Since there must exist one language of which it is true, the set of languages that satisfy it isn't empty. Since there must exist one language of which it is false, the set of languages that satisfy it is not exactly the set of RE languages and so we have a proper subset.

So, to use Form 2 of Rice's Theorem requires that we:

0) Specify some set  $C$  (by specifying a membership predicate  $P$  or some other way).

1) Show that  $C$  is a subset of the set of RE languages (which is equivalent to saying that the domain of its membership predicate is the set of RE languages)

2) Show that  $C$  is a proper nonempty subset of the recursive languages by showing that

a)  $C \neq \emptyset$  (i.e., its characteristic function  $P$  is not trivially false), and

b)  $C \neq RE$  (i.e., its characteristic function  $P$  is not trivially true).